University of Alberta

VALUING SOFTWARE SERVICES: THE REAL OPTIONS-BASED MODULARITY ANALYSIS FRAMEWORK

by

Brendan Michael Tansey

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2008

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-45893-8
Our file Notre référence
ISBN: 978-0-494-45893-8

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

As the demands of a market change, software services can rapidly evolve to suit emerging conditions. Yet rapid evolution risks producing hasty, suboptimal, decisions: decisions that, if optimized, could maximize resource efficiency and profit for the business. Leveraging a structured framework for evaluating possible evolution alternatives would reduce the quantity of income lost to uninformed choices. To this end, I describe the Real Options-based Modularity Analysis (ROMAN) Framework for evaluating the net value of a prospective service. ROMAN combines traditional software valuation with flexibility analysis, using real option theory, to produce a value estimate for a service. To make this process more accessible to practitioners, I also implement the framework in a decision support system that can be used to inform the decision-making process. I also perform a study to determine default maintenance parameters for the COCOMO II cost estimation model using a publicly available set of open source applications.

Table of Contents

1 Introduction **1**
- 1.1 The Problem of Valuation . . . 1
- 1.2 Why Study Software Services? . . . 3
  - 1.2.1 A Service Economy . . . 3
  - 1.2.2 Software-based Service Oriented Architecture . . . 4
- 1.3 Approach Overview . . . 7
- 1.4 Contributions of this Thesis . . . 9
- 1.5 Thesis Organisation . . . 9

2 Software Cost Estimation Models **11**
- 2.1 Size Metrics . . . 11
  - 2.1.1 Source Lines of Code . . . 12
  - 2.1.2 Function Points . . . 13
- 2.2 Cost Model Background . . . 14
- 2.3 COCOMO II.2000 Details . . . 16
  - 2.3.1 Required Inputs . . . 17
  - 2.3.2 Effort Estimation . . . 18
  - 2.3.3 Schedule Estimation . . . 21
  - 2.3.4 The COCOMO Suite of Models . . . 22
- 2.4 Summary . . . 23

3 Valuation Approaches **24**
- 3.1 Project Valuation . . . 24
  - 3.1.1 Net Present Value . . . 25
  - 3.1.2 Internal Rate of Return . . . 26
- 3.2 Real Options Primer . . . 27
- 3.3 Real Option Valuation . . . 28
  - 3.3.1 Black-Scholes . . . 29
  - 3.3.2 Black's Approximation . . . 30
  - 3.3.3 Binomial Lattice . . . 31
- 3.4 Volatility Estimation . . . 33
- 3.5 Summary . . . 35

4 Related Work **37**
- 4.1 Software Economics . . . 37
- 4.2 Architectural Models . . . 40
- 4.3 Software Service Valuation . . . 42
- 4.4 Summary . . . 44

5 Methodology **45**
5.1 Case Study Introduction 45
5.2 ROMAN Framework 46
5.2.1 Step 1: Estimate Basic NPV of Alternatives 47
5.2.2 Step 2: Model Project Uncertainty 48
5.2.3 Step 3: Identify General Project Flexibility 49
5.2.4 Step 4: Identify Modular Flexibility 52
5.2.5 Step 5: Calculate NPV Including Flexibility 57
5.2.6 Alternative Calculation Procedure for Step 5 - Risk Neutral Valuation 59
5.3 RomanDSS - The ROMAN Framework Decision Support System . 60
5.4 Default COCOMO II Maintenance Parameters 63
5.4.1 Experiments with a Controlled CM 65
5.4.2 Experiments with a Controlled DM 67
5.5 Summary 69

6 Conclusions **71**
6.1 Summary 71
6.2 Future Work 73

Bibliography **76**

A Acronym Reference Guide **82**

List of Tables

2.1 Simple algorithmic software cost models using SLOCs [54]. 14
2.2 Simple algorithmic software cost models using FPs. 15

3.1 Input variables for value models. 36

5.1 Initial development COCOMO II input variables for MovieInfo movie information web service. 48
5.2 Projects and their statistics used in the COCOMO II default maintenance parameter study. 64
5.3 Experimental results of CM tests. 66
5.4 Experimental results of DM tests. 67

List of Figures

1.1 GDP composition of the 15 global entities with the greatest GDP, in order of descending GDP from left to right. 5
1.2 Labour force compositions of the same global entities as presented in Figure 1.1. 5

3.1 Profitability of call and put options. 28
3.2 A binomial lattice for asset valuation, as in [15]. 31
3.3 The binomial lattice for the value of a call option based on the asset in Figure 3.2, as in [15]. 31

5.1 General event lattice. 49
5.2 Event lattice for MovieInfo. 50
5.3 Profitable option exercise points. 53
5.4 Profitable option exercise points including overlapping options. . . . 58
5.5 Final decision lattice with optimal option exercise times. 59
5.6 RomanDSS execution graph. 61
5.7 IM and DM values for given values of CM. 66
5.8 IM and CM values for given values of DM. 68

Chapter 1

Introduction

1.1 The Problem of Valuation

As the number of services delivered through software infrastructures increases, so too must our understanding of these systems. Software services are unlike traditional software products: they can evolve rapidly to meet the changing needs of a market, and they can do so without the time- and money-consuming processes of installation and maintenance associated with conventional products. This leads not only to a rapid response time for the addition of required features, but also to the possibility of innovation, as services from multiple sources can be easily composed together to form a larger more functional service (such as many web application hybrids, or "mashups", seen on the Internet today). However, the primary goal of a business is to maximize profit; to achieve that goal, companies need to possess a method of determining which evolutions of their software services will produce the greatest expected value in the future. These decisions should be based on more factors than traditionally considered: along with software cost and quality attributes, value created through system flexibility needs also to be taken into account. Areas of the service that can be modified, replaced, removed, or built upon may not provide any direct income in the short term, but in the long term they can be used to alter the service to hone the functionality to consumer requirements. Not only will this allow for profit optimisation, it could also have the positive side effect of fostering the development of innovative new services; the price of initial development could be as low as that associated with combining existing services, allowing for

new offerings at a low cost. In order to determine which out of a set of possible alternative new services should be created, each prospect needs to be valuated in this manner.

Valuation is the process of assessing or estimating the net worth of an entity. In the context of modern businesses, valuation is used extensively to determine the positive or negative effect strategic decisions or new offerings will have on a company. At the highest level of abstraction, two basic components constitute net value: cash outflows (cost) and cash inflows (revenue). Calculating net value at this point is trivial; simply subtract cash outflows from inflows to obtain expected profit. However, these quantities are rarely known with certainty, especially when estimating future cash flows. In order to perform this estimation, one must predict the future positive and negative cash flows that will result from a given business operation.

To maximize accuracy, these predictions require domain-specific information; therefore, any fine-grained estimation problem needs to be constrained to a particular field. This thesis examines the estimation of net value in relation to software service projects; that is, software projects that provide functionality on an ongoing basis to the customer, usually in the form of a web service. Businesses generally generate income from this type of offering on a per-usage basis, as opposed to traditional software or licence sales. A prediction model in this context would be valuable in two primary ways. First, it would allow businesses to determine whether or not a proposed service would be profitable. As an extension of this concept, it would also permit companies to choose the most profitable out of a set of possible projects given a finite resource set, improving resource consumption efficiency. Second, the model could inform a business on the profitability of evolving an existing project to meet additional customer demands or desires. Again, this would also enable developers to make informed choices between multiple evolution scenarios for an existing service in order to maximize profit.

In addition to traditional estimation components, i.e., most likely costs and revenues, services create large amounts of flexibility for their owners. As market demands shift, developers can add or remove modules from a service to meet current

requirements, or new services can be created through an amalgamation of smaller services to facilitate rapid development. This flexibility has value, but this value is often hard to quantify, which leads to many interesting questions. How much does flexibility add to the value of a project? Will designing a project with the intention of providing flexibility generate a positive return on investment? Will it increase profits? When faced with the choice of developing one out of a set of possible projects or project evolutions, each of which has its own flexible value, these questions require answers in order to maximize expected value. To fully investigate these questions, we require a complete financial model to account for both costs and sources of revenue, including the value of flexibility, and this model needs to be domain-specific to software services.

1.2 Why Study Software Services?

This question has two parts: the general question of "why study services at all?", and the more direct "why study services that are implemented in software?" In essence, the answer to both of these questions is that we can't afford not to, as both services in general and software-based services in particular are where industry is migrating. This section provides details of this migration and the overall importance of services.

1.2.1 A Service Economy

Services in general are the single largest aspect of the world's economy. As of 2007, services generate 64% of the world's GDP[1], industry creates 32%, and agriculture produces a scant 4% [2]. Figure 1.1, which shows the GDP composition of the top 15 global entities, clearly reveals that services are the dominant economic contributor in all cases, save for China. In contrast, services account for only 38.8% of the world's labour force, with industry representing 20.5% and agriculture the majority at 40.7%. Figure 1.2 shows the same entities as Figure 1.1, but provides labour force breakdowns.

[1]GDP based on PPP (purchasing power parity) figures, not OER (official exchange rates), as PPP is a better representation of a society's effective wealth.

Notably, the graph for all but four entities is quite similar, with the majority of the workforce of the top countries by GDP employed in services. The four oddities are Brazil, India, China, and the world as a whole. The latter is a sobering fact, as the vast majority of agricultural-based economies are also among the poorest in the world[2]. However, among the major world economies, there has been a recent push towards a greater and greater portion of the labour force in services. In the last 25 years Brazil has seen a 20% growth in its service labour force; similarly, India has increased 28% and China 191% [53].

Clearly, services are increasingly becoming the dominant feature of the economy. As further evidence, many business models are shifting from the traditional goods-oriented sale model to a continuing service-oriented usage or licensing model. Telephone companies commonly offer free or discounted cellular phones with a service agreement to use that phone. Movie rental companies such as Zip.ca offer plans based not on the number of movies you rent, but on your desired level of service (i.e., the number of movies you can have checked out at any one time). Many massively multi-player online games such as World of Warcraft and Eve Online are either free to acquire through download or sold as trial versions extremely cheaply, but require the user to pay a monthly fee to use the online environment on a continuing basis. It is this pervasive style of business model, concentrated in the realm of software, that is relevant to this thesis.

1.2.2 Software-based Service Oriented Architecture

In the general sense, the concept of a Service Oriented Architecture (SOA) is not limited solely to software, but is "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains" [52]. The idea behind SOA is satisfying the needs of an entity that it cannot meet itself by matching these needs with the capabilities offered by other entities. This is achieved with the three key concepts of SOA: visibility, interaction, and real world effects. Capabilities offered by an entity are visible to other interested entities; an entity

[2]Burundi: agriculture = 93.6% of labour, GDP = 5.8B; Niger: agriculture = 90% of labour, GDP = 12.3B; Rwanda: agriculture = 90% of labour, GDP = 13.7B; Burkina Faso: agriculture = 90% of labour, GDP = 18.9B; Malawi: agriculture = 90% of labour, GDP = 8.3B [2, 58]

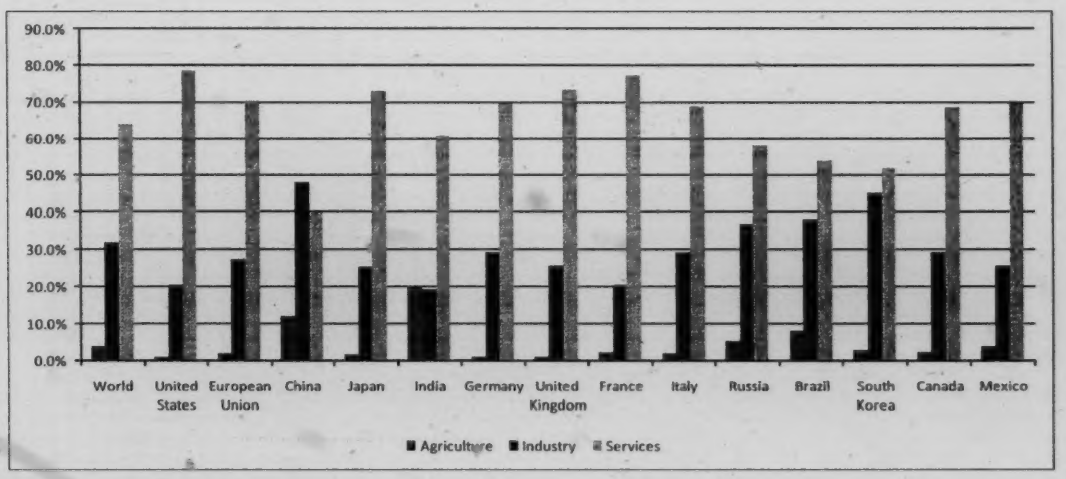

Figure 1.1: GDP composition of the 15 global entities with the greatest GDP, in order of descending GDP from left to right.

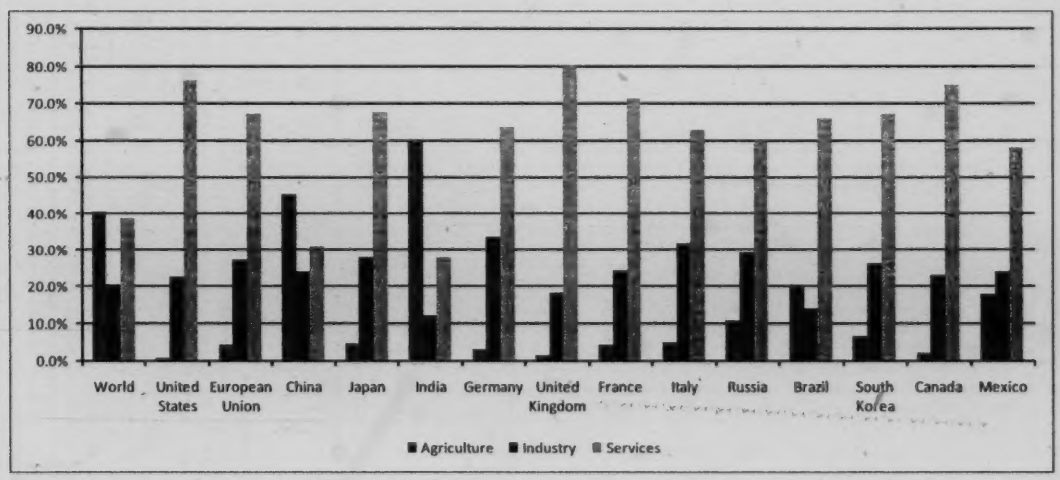

Figure 1.2: Labour force compositions of the same global entities as presented in Figure 1.1.

with needs matching a visible capability will interact with the offering entity to make use of that capability; this interaction causes real-world effects, which satisfy the requirements of the needy entity. Services facilitate this process: a service description allows capabilities to be visible to needs, describes the necessities for communication such as required inputs and expected outputs, and lists what effect the capability will have upon successful invocation.

Software-based SOA implements these concepts with software services, which are most commonly presented as web services. Numerous technologies have been created to handle every aspect of software SOA implementation. UDDI (Universal Description, Discovery, and Integration) and WSIL (Web Service Inspection Language) are used for visibility by listing the capabilities of services in a searchable directory. Interaction is handled by SOAP[3], which is an XML-based messaging protocol for communication between web services. The effects, inputs, and outputs of services are described in a WSDL (Web Service Description Language) document. Also of particular interest is BPEL (Business Process Execution Language), which can be used to create service compositions in order to produce a service based on numerous sub-services. These aggregations create a vast amount of flexibility when designing and maintaining services; at any time, new components can be added, extraneous components can be removed, and outdated components can be replaced or upgraded as the market demands. This complexity is a major contributing factor to the problem of service valuation.

In 2005, the technology research and advisory company Gartner published a report containing three interesting predictions: "by 2008, most application software revenue will come from products that were built using SOA", "by 2008, SOA will provide the basis for 80 percent of development projects", and "by 2010, 80 percent of application software revenue growth, including licenses and subscription fees, will come from products based on SOA" [22]. Consultants Forrester Research provide similar assessments, and in 2006 stated that "53% of enterprises are using SOA now or will use it by the end of 2006" [37]. In 2007 they updated this

[3]SOAP was originally an acronym for Simple Object Access Protocol, but was renamed to simply SOAP by the XML Protocol Working Group. See `http://www.w3.org/2000/xp/Group/1/06/f2f-pminutes`

information, stating "62% of enterprises are using or will use SOA by the end of 2007" [38], implying that the industry is well on its way to a major SOA integration. With this quantity of industry interest in software-based SOA, research into applicable valuation frameworks is imperative.

1.3 Approach Overview

To assist with this problem, I introduce the Real Options-based Modularity ANalysis (ROMAN) framework. This work extends the general valuation framework of McKinsey [55], and tailors it to specifically apply to software services. At the core of this framework is real option analysis, which can place a numeric value on intangibles such as flexibility. An *option* is the right, but not the obligation, to purchase or sell an asset at a fixed price for a limited time [28]. This fixed price is referred to as the 'strike price', while the current market value of the asset is referred to as the 'spot price'. A *real option* is a refinement of this definition that only deals with real assets, not financial derivatives. In this context, the strike price is no longer the exercise price for the option, but the development cost of the action associated with embedding the option in the project. Similarly, the spot price is not the current value of the stock, but the total additional revenue afforded by the option. There are two basic categories of options: 'call' and 'put' options. With a call option, the owner has the right to acquire an asset for a fixed price, and conversely a put option gives the owner the right to sell an asset for a fixed price. Various styles of options exist, such as European options, which can only be exercised at the date of expiration, and American options, which can be exercised prior to expiration. All of the options discussed in this thesis will be treated as American options.

Multiple methods exist for placing a value on such options; popular models include Black-Scholes [17] and the binomial model [24]. In this thesis I use the binomial model, as it allows for early execution of options, can easily handle multiple interacting options, and provides the user with a clearer idea of possible future states for the project.

ROMAN also assists the user in identifying locations where real options could

exist or be intentionally embedded in a project. These can be either at the general project level, or at the design level. I relate options at the design level to the six modular operators developed by Baldwin and Clark [13]: *splitting* a system into sub-modules, *substituting* one module for another, *augmenting* the system with a new module, *excluding* a module from the system, *inverting* to create new design rules and standard components, and *porting* a module to another system. The framework spurs the valuator to question how each operator can provide flexibility to each proposed system under consideration, based on currently available information. This flexibility is then related to real options for the purposes of numerical assessment.

As the overall goal of the ROMAN framework is to facilitate comparison between alternative future services, the final stage of the framework involves estimating the net present value (NPV) of each alternative system, including the value provided by flexibility. The costs for creating each system are analyzed at this stage as well; this task is performed by the COCOMO II cost estimation model [19]. With these values calculated, the concerned party can choose to develop the service (or services) which will produce for them the greatest future return on their investment.

As many of the calculations involved in determining the final NPV of a system are complex, given that some inputs are not known for certain and can be probabilistic distributions, I have developed RomanDSS, a decision support system, to assist with the calculation process. This DSS also automates some of the steps in the framework, making the process more accessible to interested parties. Uncertain input parameters are modeled as distributions, which are evaluated with Monte Carlo simulation [70].

In essence, the ROMAN framework provides a structured approach to software service valuation. For the predicted value of possible future projects to be comparable with each other, the same process must be taken to valuate each. ROMAN presents a set of steps to undertake, complete with details of how these steps could apply to a project, and uses cost and value models to present a comparable NPV for each alternative.

1.4 Contributions of this Thesis

This thesis makes three main contributions to the field:

1. The first contribution is a complete net valuation framework. The ROMAN framework is complete in the sense that it incorporates both the cost of creating software and the value that owning the software will likely produce for its creator. The framework also contains guidelines for its systematic application to a proposed software service. While related work has been concerned with architectural quality [59] and architectural stability [9], this work focuses on architectural profitability.

2. The second major contribution is the RomanDSS tool, which allows its users to quickly and easily analyze a proposed system under the ROMAN framework. Performing NPV calculations with probabilistic inputs requires an automated system, and the Monte Carlo simulation abilities of RomanDSS provide just that. As an added benefit, the DSS allows users to estimate the value of a project without requiring knowledge of the underlying models or processes.

3. Finally, the third contribution stems from development of RomanDSS. In an attempt to simplify the set of inputs to the system I performed research into default values for common inputs. Notably, given the percent of code modified in a system, I calculate the average level of design modification that takes place (i.e., the percentage of methods changed) and the average retest impact of that change (which is the percentage of a system that has to be retested if a section of code is altered).

1.5 Thesis Organisation

Each stage of this research is discussed in more detail herein. As such, the remainder of this thesis is organized as follows. Chapter 2 provides background information on software cost models and provides details of the model that this thesis uses. To complete the economic calculations, Chapter 3 gives background on valuation

techniques, both general and real options based. Chapter 4 discusses other research in this field, including related studies of this problem. Chapter 5 describes the steps of the ROMAN framework and the associated DSS, as well as provides an example case study to help illustrate the process. This chapter also contains the experimental results of my study on default COCOMO II input parameters which can be used in the framework. Chapter 6 summarizes the important points of this research and outlines a future research agenda for this area.

Chapter 2

Software Cost Estimation Models

Cost prediction is one of the two cornerstones of net value estimation. Fortunately, this subject area has been researched for roughly three decades, with significant progress being made in the late 70s and early 80s. This section provides an overview of this research, including an in-depth explanation of the model that is primarily used for calculating cost in this thesis, COCOMO II. However, prior to discussing cost models, this section provides background information on a major input into all cost models: the size of the service oriented application to be developed, which relies on software size metrics.

2.1 Size Metrics

The cost of developing a piece of software, quite obviously, depends heavily on the size of the project to be undertaken. All other factors constant, a larger project will cost more to develop than a smaller project, both in terms of time and capital.

There are two main size metrics that are generally applied to software development projects: source lines of code (SLOCs) and function points (FPs). Each of these has its own benefits and weaknesses, which are described herein. Other size methods have been proposed over the years, such as judging the size of the project by the number of files it contains [39] and offshoots of FPs such as Object Oriented FPs [6], but no other metrics have reached the widespread usage level of SLOCs and FPs.

2.1.1 Source Lines of Code

Source lines of code, or SLOCs, are perhaps the simplest of the popular size metrics. As an evolution from delivered executable machine instructions (DEMI) and delivered source instructions (DSI) [18], SLOCs measure the number of instructions present in a collection of code. As the amount of code produced to perform a task can vary greatly between programming styles and individual programmers, SLOCs are categorized as either physical or logical. Physical SLOCs are a simple count of the actual lines of text in a source code document; this measure is rarely, if ever, used as a size metric in modern estimation systems due to its high variance. Logical SLOCs are lines of program functionality, composed of executable statements, variable and function declarations, and compiler directives that are not automatically generated. Each logical SLOC can be composed of multiple physical SLOCs, and one physical SLOC can contain multiple logical SLOCs. In 1992 the Software Engineering Institute published a checklist for counting logical SLOCs in source code [60], which has become the standard method of determining logical SLOCs. This set of rules has been adopted by COCOMO II as the basis of its sizing algorithm [19].

Developers have written many tools to count SLOCs, both physical and logical. CodeCount [72] is quite popular, and counts logical lines of code in adherence to the SEI standards. Its main weakness is that, in its current revision, it cannot handle source files written in Java 5. LOCC [25] handles modern languages such as Java 5, but does not adhere to SEI standards for logical SLOC counting. However, as LOCC is open source, with minor modifications it can be adapted to achieve this goal. This thesis uses just that: a version of LOCC modified to a) count logical SLOCs according to the SEI standard, and b) count SLOCs on a per-method basis instead of the default per-class basis.

Though the use of logical SLOCs as a size measure is commonplace and fairly intuitive, it is not without problems. While the tools mentioned above make counting SLOCs in existing source code nearly trivial, SLOCs are quite difficult to estimate early on in the lifecycle of a software project, which is when predictions are most important [7]. There are other counting methods that are less affected by this

shortcoming, such as function points.

2.1.2 Function Points

Function points (FPs), invented by A.J. Albrecht in 1983 [4], are composed of five factors: external input types, external output types, logical internal file types, external interface types, and external inquiry types. External input and output types are interactions with the user that result in a change of a logical internal file or the production of a report. Logical internal file types are logical collections of user data or control information. External interface file types are files shared between applications, and external inquiry types are user requests that generate an immediate output [51]. Since these concepts are higher level than their source code implementations, they can be determined earlier in the development cycle, such as at the requirements stage. This is a significant advantage over using SLOCs, as estimations should optimally be performed as early as possible to reduce unprofitable development.

FPs are not without problems either though. They are significantly less straightforward to count than SLOCs, and are open to more subjectivity on the part of the counter [47]. Not all function points are created equal either: developing a function point in a lower-level language such as assembly will take considerably more effort than creating a function point in a higher-level language such as BPEL. However, while there are cost estimation models that natively handle FPs as input [26], all of these models are either simple formulas with constants based on a limited training set size, or are only applicable to specific languages such as C and COBOL [54]. FPs are generally most useful in estimation models when converted back to an equivalent SLOC count, using a conversion ratio specific to the language used for the project. The company SPR maintains an updated list of conversion ratios for modern programming languages, and currently lists over 500 languages [66].

In principle, one should calibrate this FP-to-SLOC ratio in the context of their own development environment to obtain the most accurate estimate of effort. This can be done by using a method known as "backfiring", which involves counting the function points of a previously completed piece of code developed in the same en-

vironment and comparing this count to the logical lines of code of that project [44]. Recently, Aggarwal *et al.* have researched estimating SLOCs of a project using neural networks, with FPs and programming language as inputs. The results of this work look quite promising, yielding a mean absolute percentage error of 14.65 using Bayesian regularization [3].

Since FPs were first defined they have undergone many refinements, with the evolution currently being controlled by the International Function Point Users Group (IFPUG). The most recent counting practices manual, as of the time of this writing, was published in January 2005 [41].

2.2 Cost Model Background

Barry Boehm, in his seminal book *Software Engineering Economics*, defines seven techniques for estimating software development cost [18]. The first of these categories is algorithmic models: models that predict cost based on a set of input parameters, or "cost drivers". Most calculable cost models fit into this category since the concept is quite intuitive. Of these, the simplest models use project size as their sole input; Table 2.1 presents models that use SLOCs as input, and Table 2.2 lists models that use an FP count as input. These models can be developed through experimentation, mathematical analysis, or machine learning techniques applied to data from previous projects. The measure of effort provided by these models is in person-months.

$Effort = 5.2(KSLOC)^{0.91}$	Walston-Felix model
$Effort = 5.5 + 0.73(KSLOC)^{1.16}$	Bailey-Basili model
$Effort = 5.288(KSLOC)^{1.047}, KSLOC > 9$	Doty model
$Effort = 3.2(KSLOC)^{1.05}$	Boehm simple model
$Effort = 3.0(KSLOC)^{1.12}$	Boehm average model
$Effort = 2.8(KSLOC)^{1.20}$	Boehm complex model

Table 2.1: Simple algorithmic software cost models using SLOCs [54].

Not all algorithmic models are as simple as those in Table 2.1 and Table 2.2 though; for example, Putnam's SLIM model takes into account schedule compression [63] (managers attempting to meet a deadline that comes before the time at

$Effort = -13.39 + 0.0545(FP)$	Albrecht-Gaffney model [5]
$Effort = 60.62 + 7.728 \times 10^{-8}(FP)^3$	Kemerer model [26, 46, 54]
$Effort = 585.7 + 15.12(FP)$	Matson *et al.* model [54]

Table 2.2: Simple algorithmic software cost models using FPs.

which the project would normally be complete). However, almost all complex algorithmic models are either proprietary or have become antiquated. SLIM is now owned by QSM [64]. The PRICE S model, created by Putnam, has evolved into True S and is owned by PRICE Systems [35, 62]. SPQR/Checkpoint is now SPR's KnowledgePLAN software tool [43, 67], Estimacs is owned by Computer Associates International [65, 23] but has not been recently updated, and Galorath Incorporated owns SEER [42, 36]. COCOMO is the notable exception to this list - originally proposed by Barry Boehm in 1981 [18], it has been extensively published, evolving into COCOMO II in 1995 [21] and COCOMO II.2000 in 2000 [19]. COCOMO II is easily accessible, can be fine-tuned to suit almost any development environment and project size, and natively handles code reuse. These qualities lend it well for use in this research, and as such COCOMO II will perform the cost estimation tasks of this work.

The second largest category of cost estimation techniques is that of expert opinion, or expert judgment. This method utilizes experts who are experienced in estimating the costs of projects similar to the one being analyzed. Expert judgment is very versatile since many nuances of a project that can be overlooked by an algorithmic model can be taken into account, but it can also be subject to bias on the part of the estimator. More importantly to many organizations requiring estimation, qualified estimators are not always available. It is for this reason that expert opinion estimates are not used in this research.

The remaining five estimation techniques are estimation by analogy, Parkinsonian estimation, price-to-win estimating, top-down estimating, and bottom-up estimating, all of which are fairly straightforward. Estimation by analogy uses data from completed projects that are similar to the one under analysis, assuming that the costs will be correlated. This method benefits from past experience of like

projects, but relies on the basis projects far too heavily; if similar projects are not as comparable as assumed, or if similar projects do not exist, the estimation will be quite inaccurate or will be impossible to perform. Parkinsonian estimation assumes that the cost of the project will be equal to the budget allocated for the project, whether or not the entire budget is required. A similarly terrible idea, price-to-win estimating involves pricing a software project at a level that is preferable to clients or management, but is usually highly unrealistic, in order to win development contracts. Top-down and bottom-up estimating use the other estimation methods: top-down estimation predicts the cost of the entire system and divides that cost over the component parts; bottom-up estimation predicts the cost of each component, the summation of which is the estimated cost for the system. Software services are created by a composition of parts, and as such bottom-up estimation is directly applicable when used in conjunction with the aforementioned COCOMO II model.

2.3 COCOMO II.2000 Details

Other than estimating the cost of creating a project from scratch, one of the main benefits of COCOMO II is the ability to account for code reuse. In this respect, COCOMO II estimates the cost, in terms of programmer effort, of creating a software development project as a function of the transformational complexity of the application from an existing state to its envisioned state. All costs are given as both the number of Person-Months (PM) and the time in calendar months required to perform the change. Depending on the current stage of development, COCOMO II contains slightly different models for cost calculation, namely early design, post-architecture, or maintenance models.

For optimal performance, a number of COCOMO II parameters must be calibrated to suit the specific software development environment. COCOMO II.2000 comes with a set of default constants in place (A, B, C, and D in the equations below), determined through testing the model on 161 sample projects. While these parameters are adequate as a starting point, they will likely produce less accurate results than if the model were fully calibrated based on the development environment,

the development team, and similar projects undertaken in the past. Section 7 of the model definition manual discusses how this calibration can be performed [19]. Other than the recommended calibration, a number of variables are required as inputs into the COCOMO II estimation model, depending on the maturity of the project and the existence of code reuse. This section will discuss these necessary inputs, followed by details of the COCOMO II models and a discussion of other models that are based on COCOMO.

2.3.1 Required Inputs

If the project in question does not employ code reuse, the sole required input into COCOMO is the amount of code to be written, in terms of thousands of SLOCs. Code reuse adds an additional four required input parameters.

Size of code base: This input variable represents the size of the original code base to which the adaptations are being made. Some section of this code is directly modified to suit the needs of the new service. This quantity is accepted in thousands of SLOCs, and is referred to as *adapted* $KSLOC$ in the COCOMO II model manual [19].

Amount of code added: This is code added that is not from the original code base. This can include "glue code", or code to introduce sections of completely new functionality. If the new project is completely new development, all of the other input values can be zero, in which case *code added* represents the entire size of the development. In the COCOMO II model manual this quantity is referred to as *new* $KSLOC$.

Amount of code modified, or percent of code modification (CM): This represents the amount of code from the original code base that has been directly modified to accommodate the new desired functionality. CM represents this quantity as a percentage, equal to $(code\ modified)/(code\ base)$ (if *code base* is zero, CM is taken to be zero).

Percent of design modification (DM): This parameter denotes the percentage of the overall design of the adapted code that requires modification during the process of creating the new service. According to the COCOMO II.2000 model manual,

this quantity is necessarily subjective. However, a study by Robert Leitch defined this quantity as the percentage of methods in source code that are modified during reuse [49]. For all instances where I calculate DM in this thesis, this is the method that I use.

Retest impact (IM): This is the percentage of integration and testing effort required for performing the modification. For simplicity, this value can be the percentage of the original code base that needs to be retested. Note that this value can exceed 100% if more code requires testing than was originally in the code base. In Section 5.4 I discuss predicting this value based on a measure of CM or DM.

2.3.2 Effort Estimation

COCOMO II.2000 claims an effort estimation accuracy of within 30% of the actual value 75% of the time. With calibration, this accuracy increases to within 30% of the actuals 80% of the time [20].

The first stage of estimation is determining the size of the project. Using the inputs discussed in the previous section, the formula for this is shown in Equation 2.1.

$$Size = \left(1 + \frac{REVL}{100}\right) \times (New_KSLOC + Equivalent_KSLOC) \quad (2.1)$$

$$Equivalent_KSLOC = Adapted_KSLOC \times \left(1 - \frac{AT}{100}\right) \times AAM$$

$$AAM = \begin{cases} \frac{[AA+AAF(1+(0.02\times SU\times UNFM))]}{100}, \text{ for } AAF \leq 50 \\ \frac{[AA+AAF+(SU\times UNFM)]}{100}, \text{ for } AAF > 50 \end{cases}$$

$$AAF = (0.4 \times DM) + (0.3 \times CM) + (0.3 \times IM)$$

Performing these calculations will provide $Equivalent_KSLOC$ ($EKSLOC$), i.e., the number of *logical* source lines of code (SLOC) divided by 1000, required to create the new service. This value is a measure of effort, not necessarily an exact count of actual code lines to be written; it represents the effort required if the equivalent number of lines of code would in fact have to be written. Essentially, it reflects the effort that the "adaptation" of the existing service to the new one will require. This value can then be added to New_KSLOC, additional code that needs

to be added to the system, and adjusted by $REVL$, the requirements evolution and volatility level, which is the percentage of code that is discarded from the original module during the modification. This process provides the user with $Size$, a measure of the system size, accounting for reused code.

The calculation of $EKSLOC$ relies on a number of parameters. AAM is the Adaptation Adjustment Modifier, and AAF is the Adaptation Adjustment Factor; they represent the effort of fitting the adapted code to an existing product, and the relative size of the modification, respectively. AT is the percentage of code that can be translated using automatic translation tools from the existing system to the new one, thereby not requiring any effort. AA is the assessment and assimilation increment, which is a measure of the effort required to determine if the old code base is suitable for the new project. SU is the level of software understanding increment that models the quality and complexity of the old code base. If CM and DM are both equal to zero, then SU is also set to zero. This is because if no changes to the code are necessary, then programmer understanding of the code is irrelevant. $UNFM$ is the familiarity of the programming team with the code that they are modifying. AA, SU, and $UNFM$ are project-specific parameters that need to be set using the calibration guidelines in the COCOMO II.2000 model manual.

In order to translate $Size$ into a useful effort measure, either the Early Design or Post-Architecture model must be used. The choice of model depends on the current stage of development of the project at the time that the estimation is to take place. As a guideline for choosing one model over the other, the COCOMO II model manual gives the following explanation:

> The Post-Architecture is a detailed model that is used once the project is ready to develop and sustain a fielded system. The system should have a life-cycle architecture package, which provides detailed information on cost driver inputs, and enables more accurate cost estimates. The Early Design model is a high-level model that is used to explore of architectural alternatives or incremental development strategies. This level of detail is consistent with the general level of information available and the general level of estimation accuracy needed. [19]

The Post-Architecture model is shown in Equation 2.2.

$$PM = A \times Size^E \times \prod_{i=1}^{17} EM_i \tag{2.2}$$

$$E = B + 0.01 * \sum_{j=1}^{5} SF_j$$

This model estimates the predicted Person-Months (PMs), in programmer time, that will be required for the entire translation process. PM relies on a number of other variables. A is the effort coefficient and B is the scaling base-exponent, both of which have default values: $A = 2.94$ and $B = 0.91$. However, for optimal results these values should be calibrated to reflect the development environment. There are 17 EM_is, or Effort Multipliers, to control the rate at which various factors of the software development process affect the effort required to perform a change. E is the sizing scale exponent, which takes into account the five SF_js, or Scale Factors, which determine the effect of economies or diseconomies of scale on the software project. The intuition behind E is that if factors exist that lead to diseconomies of scale, such as a team that only interacts internally with great difficulty or conflict, then the effort required to produce a line of code will be multiplicatively greater than the effort required for a seamlessly interacting team.

Equation 2.3 demonstrates the Early Design model, which involves a similar calculation, but uses a different set of effort multipliers. The equation for E is the same as in Equation 2.2.

$$PM = A \times Size^E \times \prod_{i=1}^{7} EM_i \tag{2.3}$$

The reduction in the effort multiplier set stems from the lack of information that is available early on in the development process. For a description of the scale factors and effort multipliers, please refer to Sections 3.1 and 3.2 of the COCOMO II model definition manual, respectively [19].

2.3.3 Schedule Estimation

The time required to perform a change, from the requirements determination stage to the requirements satisfaction verification stage in the waterfall development models, can also be calculated using COCOMO II. The accuracy of this calculation is slightly less than that of effort estimation: within 30% of the actual value 64% of the time without calibration, and within 30% of the actuals 75% of the time with calibration [20]. The equations for this estimation is as follows in Equations 2.4 and 2.5.

$$TDEV = \left[C \times (PM_{NS})^F\right] \times \frac{SCED\%}{100} \tag{2.4}$$

$$F = (D + 0.2 \times [E - B]) \tag{2.5}$$

$TDEV$ is the time required to perform the required changes, measured in calendar months. B, C, D, and E are environment-specific coefficients. E corresponds to the value for E from the effort calculation; B, C, and D are unique to the $TDEV$ equation and should be calibrated, though they have default values of $B = 0.91$, $C = 3.67$, and $D = 0.28$. PM_{NS} is the nominal effort involved in making the required changes; it is measured in person-months, and does not take the $SCED$ parameter into account. $SCED$ is the required schedule compression, ranging from 1.43 for an accelerated schedule to 1.00 for a drawn-out schedule.

Once the effort, measured in person-months, has been determined, it can be directly translated into a monetary cost by simply multiplying it by the monthly salary of the programmers involved in the project. This identical procedure, including use of the code reuse model, can be used to determine the cost of maintaining a service as well; the pre-maintenance version of the service can correspond to the reusable version, and the post-maintenance version can correspond to the final product in the above model. The difference between an initial reuse model and the maintenance model is that the scope of the change is usually fairly limited during maintenance. This assumption is not necessarily true for strict reuse, implying that the values for $Adapted_KSLOC$, DM, CM, and likely IM will be lower during maintenance. For maintenance, the input parameter set is also slightly reduced, as the EMs Re-

quired Development Schedule (SCED) and Developed for Reusability (RUSE) are not included in the calculation. Section 5 of the COCOMO model definition manual describes the maintenance model in detail [19].

2.3.4 The COCOMO Suite of Models

Numerous other cost models have evolved from COCOMO, forming the COCOMO suite of models. While many of these are still in research stages, four stand out as potentially applicable to the problem of cost estimation for software services. The first of these is COCOTS, the Constructive COTS (Commercial Off the Shelf) model. Since services can be created from existing software, such as a composition of other services with "glue code" added to form the composition, there may be some relevance in using a COTS-based model. However, COCOTS has not been extensively refined, and as a result is significantly less accurate than even uncalibrated COCOMO II. COCOTS can predict effort within 50% of the actual effort value 62% of the time, and within 33% of the actual effort only a miniscule 38% of the time. The accuracy values for schedule prediction are similar, with the estimates coming within 31% of the actual schedule values 54% of the time [1].

CORADMO, or the Constructive Rapid Application Development Model, is an extension to COCOMO II that adds support for projects developed under the concept of rapid application development. Software services can be small applications that are developed in a fairly short period of time, and as such a RAD model may be best suited to the task of estimating their cost. However, as promising as the idea of CORADMO looks, the last presentation related to the model, as of the date of this writing, was in May 2002 [1]. Prior to this date, insufficient research was performed to gauge the accuracy of the model, and as such it cannot be used in this work.

The other COCOMO-based model that appears to have relevance to the situation of software services is COSOSIMO, the Constructive Cost Model for Software Intensive Systems-of-Systems. Services, in their composed form, can be construed as systems-of-systems, where the composition is the main system and the compo-

[1] See http://sunset.usc.edu/research/CORADMO/coradmo_main.html#Papers, last accessed Oct. 25^{th}, 2007

nent services are the subsystems. The concept behind COSOSIMO is that each of the subsystems have a cost to develop, which can be estimated by a COCOMO-based model depending on its type. There is also a cost associated with creating the composition, the actual system-of-systems. These separate costs are combined in COSOSIMO to provide a cost for the entire system. However, like CORADMO, little has been published on this model, including accuracy statistics or a refined and calibrated algorithm [48].

While the COCOMO suite contains additional models, they are intended for large-scale systems or are concerned with aspects of software development unrelated to this work. As such, a discussion of them will not be included herein.

2.4 Summary

Significant research has been done on cost estimation models for software projects in the past. Regarding inputs, all of these models have one common requirement: the size of the project to be undertaken. Source lines of code are the most intuitive size measure of an application, but are difficult to estimate early on in a project's lifecycle. Function points, although not perfect in their own respect, provide a solution to this problem, since they can be measured by analyzing the requirements presented for the project in question. This size can then be input into a cost model for estimation purposes.

From a practical standpoint, i.e., only including models that are publicly available, COCOMO II.2000 is the front-runner of these cost estimation models: it has been extensively tested, refined, analyzed, and utilized. The models based on COCOMO that appear to be applicable to this research, namely COCOTS, CORADMO, and COSOSIMO, are not at a level of maturity sufficient enough to entrust the cost estimation of a service to them at this time.

Chapter 3

Valuation Approaches

Valuation, in this context, refers to the generic net value formulation for a project: given a set of costs and a set of revenues delineated by the period in which they are incurred or gained, net value formulae will quantify the overall financial impact the project will have on its owner. Many methods of representing this quantity exist, the most common of which will be discussed here. Methods for estimating the cost inputs for a software service project were presented in Chapter 2; the other required input, expected revenue, will be discussed in this chapter. Actual income received from providing the service to customers will likely provide the largest portion of revenue, but inherent flexibility of a service will indirectly contribute to this revenue. Income can be garnered in many project- and company-specific ways, and as such is left to individual organisations to determine. The value of flexibility, on the other hand, can be estimated using real option theory. Methods for calculating both project value and the benefits of flexibility are discussed in this chapter, along with background information on the concepts described.

3.1 Project Valuation

The worth of a project, in the most basic form, is effectively all of the revenue that the project will create, with all of the costs related to creating and maintaining the project subtracted from this quantity. However, the value of money is not constant over time; money now is generally worth more than an equal amount of money in the future. Therefore, in determining the actual value of a project, one needs to

account for the time value of money. The discounting of future monies is encompassed by the theory of Discounted Cash Flow (DCF), which is the general term for a valuation method that devalues the value of future cash flows to present day terms. This section will discuss common techniques related to DCF for valuating future cash flows: namely net present value, internal rate of return, and modified internal rate of return.

3.1.1 Net Present Value

Net Present Value (NPV) takes into account the time value of money by discounting future cash flows by a "discount rate" (r) to express them in terms comparable to the currency at $t = 0$. The general formula for calculating NPV is given in Equation 3.1, where I_t is the income at period t, C_t is the costs or expenditures at t, r_t is the discount rate at t, and C_0 represents the initial expenditures required to undertake the proposed project. If there are any cash inflows at $t = 0$, they can be directly added to NPV as well.

$$NPV = \sum_{t=1}^{n} \frac{I_t - C_t}{(1 + r_t)^t} - C_0 \tag{3.1}$$

In order to calculate NPV, these input variables need to be estimated. I_t, the predicted income of the project, can be approximated using traditional service-valuation market research such as studies of competing services, customer questionnaires, or pilot projects. This step is left up to the organisation undertaking the project. The initial cost of developing the service, C_0, can be estimated with software cost models, as were discussed in Chapter 2. If there are other initial costs associated with the project, including "soft" costs such as administrative overhead, personnel training costs, legislative lobbying costs, pilor program costs, or non-software infrastructure costs, they can be included here as well. C_t, the periodic cost, is mainly operating expenses and maintenance costs for the service. Software cost models can be used here as well. The final variable in this calculation is the discount rate, r, which is set to the rate of return the money would have realised had it been invested elsewhere. The higher this number, the more conservative the overall estimation will be. Typical values for this variable include the average rate

of return of a "tracking portfolio" of similar projects [29], the weighted average cost of capital for the project, or, for a more liberal estimation, the risk-free interest rate as determined by government bonds.

Generally, if $NPV > 0$ the project should be undertaken, as it will have a positive impact on the finances of its owner. However, the bounds of reality dictate that this cannot always be the case; if there are multiple services that can be developed, yet limited resources are available to commit to their development, only the project(s) with the greatest NPV should be undertaken. If $NPV = 0$, the project should be undertaken only if it strengthens a non-monetary variable, such as competitive position or brand recognition. Projects with $NPV < 0$ should rarely, if ever, be developed, and only then if they provide a larger indirect benefit than NPV. Section 3.3 will describe methods of placing a value on such indirect benefits.

3.1.2 Internal Rate of Return

Internal Rate of Return (IRR) is directly related to NPV; it is the discount rate (r in Equation 3.1) that will make NPV equal zero, satisfying Equation 3.2 [50]. This metric, while theoretically providing the same information as NPV, can be useful when comparing possible projects to other available investment alternatives, such as other projects or government bonds, which have their profitability figures reported solely as financial return. The general rule of project acceptance, given the choice to accept or reject a solitary project, is if $IRR > r$ (the NPV discount rate) the project should be undertaken.

$$0 = \sum_{t=0}^{n} \frac{I_t - C_t}{(1 + IRR)^t} \tag{3.2}$$

However, there is a major problem with this method of project evaluation: it assumes that IRR is constant for all t. While the NPV formula allowed for a different r_t for each t, IRR does not. Modified IRR (MIRR) compensates for this shortfall by allowing for different discount rates for each period, much like in the NPV calculation. It also takes into account the discounted cost of future expenditures that are financed from previously accumulated income versus expenditures that use capital not created by the project. The formula for MIRR satisfies Equation 3.3.

$$(1 + MIRR)^n = \frac{\sum_{t=0}^{n} (I_t - C''_t)(1 + r_t)^{n-t}}{\sum_{t=0}^{n} (C'_t)(1 + r_t)^t} \tag{3.3}$$

In this equation, C''_t represents costs at time t financed by accumulated project income and C'_t is costs at t financed by other means [50]. All other variables are the same as in Equation 3.1 or Equation 3.2

While MIRR has its uses, this work is concerned more with estimating dollar figures than rates of return. MIRR is presented here as a possible future alternative to the NPV approach, and because it is a widely accepted project valuation method; however, it is not optimally suited to this research and as such will not be discussed in greater detail.

3.2 Real Options Primer

An option is the right, not the obligation, to purchase or sell an asset for a fixed price for a limited time [28]. This fixed price is referred to as the 'strike price', while the current market value of the asset is referred to as the 'spot price'. There are two basic categories of options: 'call' and 'put' options. With a call option, the owner has the right to acquire an asset for a fixed price, and conversely a put option gives the owner the right to sell an asset for a fixed price. The profit curves for call and put options are given in Figure 3.1. Note the limited loss associated with the options. This stems from the fact that options are not required to be exercised; if exercising the option will create a net monetary loss, i.e., the option is "out of the money", it does not have to be exercised. Each option is of a specified style, which determines certain characteristics of the option; most commonly, this determines when the option can be exercised. Most financial options are one of the two most common option style: European or American. A European option can only be exercised at a single specific time: the date of expiry. American options provide more flexibility, and can be exercised at any time (that the markets are open) on or before the date of expiry. More exotic options also exist, such as Bermudan options that can be exercised only on specific days on or before expiration, but these are not necessary for this work. Note that the name of the option style has little to do with

any geographical basis for the trading of said option [40].

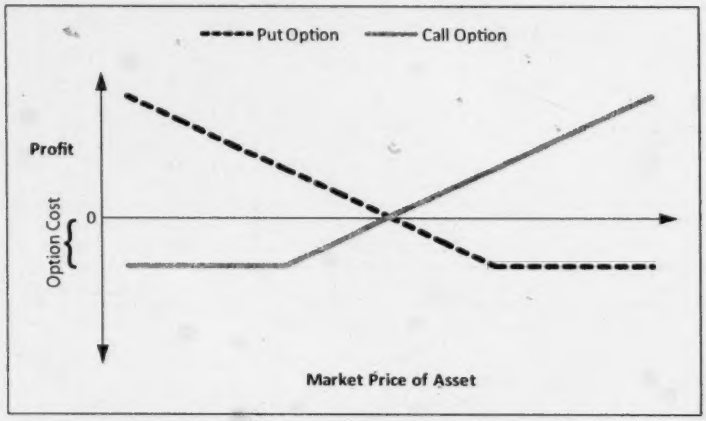

Figure 3.1: Profitability of call and put options.

A *real* option is a refinement of the option definition that only deals with real assets, not financial securities or derivatives. In this context, the strike price is no longer the exercise price for the option, but the investment cost. Similarly, the spot price is not the current value of the stock, but the total value of expected cash flow [34]. Erdogmus provides a good component by component comparison between financial options and real options in [30].

To relate service flexibility to real options, the various flexible aspects must be considered as either a put option or a call option. An example of this would be the option to upgrade a service: this is akin to a call option, where the strike price is the cost associated with upgrading, and the spot price is the benefit gained through the upgrade. Sources of real options in software services are discussed further in Section 5.2.3.

3.3 Real Option Valuation

Once project flexibility has been translated to a set of real options, those options can be evaluated and refined to reflect a quantitative monetary contribution. The conversion from concept to dollar figure requires an option valuation model or formula. The most common models for this function – Black-Scholes, Black's approxima-

tion, and the binomial tree approach – are discussed below.

3.3.1 Black-Scholes

Perhaps the most famous option pricing formula, the Black-Scholes model was developed in 1973 for pricing financial options [17, 57], for which the authors were awarded a Nobel Prize in Economics in 1997[1]. In traditional financial uses of the model, the asset on which the option is based is a non-dividend-paying stock. The key idea of the model is that a riskless portfolio of assets can be created from a combination of a position in the option and the opposite position in the underlying stock. Since both the stock and its financial derivatives (in this case the option) have the same basis for uncertainty (movements in the stock price), the price of both is perfectly correlated. If the price of the asset shifted, this shift would be offset by the opposite position in the derivative, which will shift by an equal amount. For example, to create a riskless portfolio based on Company A, one could short[2] x shares of A for $\$X$ each and buy a call option for y shares at Y each. If the market price of A's shares increases the short position will lose money, but the option will be worth more. If A loses value, the short position will be profitable, but the option will devalue. The Black-Scholes model makes use of this relationship to determine the price of an option based on the underlying stock price.

Black-Scholes has separate formulae for call and put options, which are given in Equation 3.4 and Equation 3.5 respectively. d, a variable used in both equations, is provided as Equation 3.6.

$$\Omega_{call}(U, S, T, r, \sigma) = U \cdot N(d) - S \cdot e^{-rT} \cdot N\left(d - \sigma\sqrt{T}\right) \tag{3.4}$$

$$\Omega_{put}(U, S, T, r, \sigma) = S \cdot e^{-rT} \cdot N\left(-d + \sigma\sqrt{T}\right) - U \cdot N(-d) \tag{3.5}$$

$$d = \frac{\ln\left(U/S \cdot e^{-rT}\right)}{\sigma\sqrt{T}} + \frac{1}{2}\sigma\sqrt{T} \tag{3.6}$$

[1]Robert Merton and Myron Scholes received the award; Fischer Black passed away in 1995, and the award cannot be conferred posthumously. For the pedantic, the award is not technically a Nobel Prize, but is actually The Sveriges Riksbank Prize in Economic Sciences in Memory of Alfred Nobel, as there is no prize for the field of economics described in Alfred Nobel's will.

[2]Short selling, or shorting, is defined as selling a security in which you do not own. This is a gamble that you will later be able to repurchase the security at a lower price, making a profit.

In this model, U is the spot price, S is the strike price, T is the number of periods until the option expires, r is the discount rate as discussed in Section 3.1.1, σ is the constant volatility of the asset, and $N(\cdot)$ is the standard normal cumulative distribution function. Once Ω for a real option has been determined, it can be added directly to the NPV of the project to calculate the overall value of the service.

This formula is often used as it is simple to perform the calculation with the assistance of a standard normal CDF lookup table or evaluation function that is available in most spreadsheet applications and advanced calculators. However, there are some limitations of the Black-Scholes formula. Most pertinent to this work, it can only be used to value European options, i.e., options with a single exercise date. Almost all real options can be exercised at any point up until expiry, which leads to a requirement for a valuation model for American options.

3.3.2 Black's Approximation

Black's approximation is an extension to the Black-Scholes model that allows for the pricing of American options. In effect, the approximation model performs the Black-Scholes calculation multiple times, each iteration using a different expiry date that is prior or equal to the actual expiry date, and taking the maximum value over all iterations as the option price [40]. This process is shown in Equation 3.7, where Ξ is the set of all possible exercise dates and all other variables are as described in Section 3.3.1.

$$\Omega = max(BlackScholes(U, S, \xi, r, \sigma)) \mid \xi \in \Xi,\ \xi \leq T \tag{3.7}$$

This model has limitations as well, as far as the valuation of real options related to software services is concerned: projects can embody multiple real options simultaneously, and these options interact in non-cumulative ways. The multiple options tend to conflict in certain circumstances, as is demonstrated in Section 5.2.5, and the total value is often less than a simple sum of its parts. As Black's approximation is designed to estimate the value of a single option, modelling multiple options using this method is prohibitively difficult. As such, while Black's approximation is more applicable to this research than the Black-Scholes model, neither are entirely

appropriate to satisfy the requirements of this work.

3.3.3 Binomial Lattice

The binomial option pricing formula, first proposed by Cox, Ross, and Rubenstein [24], simplifies the valuation process to a series of binary choices, where at each stage the value of the asset either increases or decreases with a certain probability. Consequently, the current value of the option increases or decreases as well. Its simplicity, and the transparency of the underlying economics, have lead to binomial models gaining widespread acceptance; binomial models are now one of the two fundamental option valuation models [15].

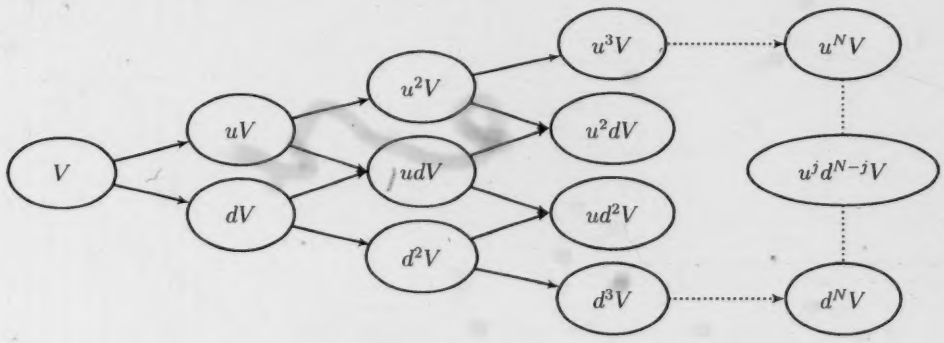

Figure 3.2: A binomial lattice for asset valuation, as in [15].

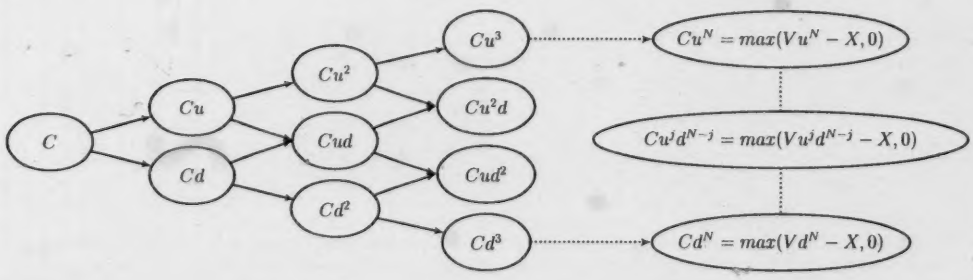

Figure 3.3: The binomial lattice for the value of a call option based on the asset in Figure 3.2, as in [15].

Figure 3.2 shows the lattice associated with the underlying asset over time, with each column representing the possible states of the asset's value at uniform intervals

of time. V is the initial value of the asset, and u and d are the upwards and downwards value multipliers, respectively. The upward or downward paths are taken with probability p and $1-p$. u and d can be calculated from the standard deviation, σ, of the underlying asset's value, as is given in Equation 3.8, where T is the time per step in the binomial lattice. Translating the value progression of the underlying asset to the value of the option produces Figure 3.3 for a call option, with X representing the exercise cost of the option. For a call option, X is the strike price of the option (S); for a put, X is the spot price (U). One important note here is that options do not have to be exercised if this would produce a net loss; effectively, the value of an option is $O = max(Y - X, 0)$, where Y is the income generated through exercising the option (the spot price of the underlying asset for a call option, or the strike price for a put option).

$$u = e^{\sigma\sqrt{T}} \quad d = \frac{1}{u} \tag{3.8}$$

To translate this procedure from a risk-free assumption to a more realistic risk-neutral environment, the probability of an increase or decrease can take into account the risk-free rate, r, discussed previously in Section 3.1.1. To do this, we set $p = (r_f - d)/(u - d)$, where $r_f = 1 + r$. Using this value, we can then determine the value of an option, as is shown in Equation 3.9. O_u is the value of the option if it increases, O_d is the value upon decrease. For a call option, $O_u = C_u = max(0, uV - X) = max(0, uV - S)$. Effectively, the complete formula for calculating the value of a call option is as in Equation 3.10, and the value of a put option is as in Equation 3.11.

$$O = \frac{pO_u + (1-p)O_d}{r_f} \tag{3.9}$$

$$C = \frac{p \times max(0, uV - S) + (1-p)max(0, dV - S)}{r_f} \tag{3.10}$$

$$P = \frac{p \times max(0, S - dV) + (1-p)max(0, S - uV)}{r_f} \tag{3.11}$$

This method of option valuation is flexible: each time period can be customised individually if so desired. It also allows for options to be exercised at any time

before expiry by allowing all intermediate values to be considered; the visibility of possible future states also allows for a rapid visual analysis of the option. These qualities make the binomial model well suited for this work; this model will form the basis for the option valuation portion of the framework presented in Chapter 5.

3.4 Volatility Estimation

Option valuation, especially in the case of real options, is not an entirely straightforward endeavour. While most input parameters to the models can be observed or estimated, volatility (σ) of V, the standard deviation of the project's returns [69], presents a certain difficulty. For projects that have yet to be undertaken, and as a result have no cash flows, it is difficult to predict how volatile these flows will be. Bajracharya *et al.* suggest a complicated process of studying the environment of existing modules in order to determine a model for calculating σ. This model would be based on factors such as "users, external services, socio-technical implications etc.", as well as the financial history of the modules [12]. While volatility is likely affected by environmental contributors, to my knowledge no researcher or practitioner has designed such a model. Poon and Granger discuss 93 papers on models that forecast volatility for financial markets; they conclude that while some classes of prediction models are superior to others, "volatility forecasting will continue to remain as a specialist subject and studied vigorously", implying that there is not a single correct solution to the problem [61]. It is also unclear how forecasting from financial markets transfers to the domain of real options.

Benaroch lists five approaches to the volatility estimation problem [16]:

1. If it is known that V will deviate by $\pm\Delta\%$, σ will simply be Δ, converted from percent to decimal form. However, this being almost the exact same situation as estimating σ itself, the same problems apply.

2. If V is composed of i cash flows, individually being V_i, an IRR can be computed for each V_i. σ can then be the standard deviation of the IRRs.

3. When the expected project revenues are of a known probability distribution

and outputs in this distribution can be produced formulaically from a set of estimable input variables, such as those summarised in Table 3.1, Monte Carlo simulation can be used to sample possible output values. The set of simulated outputs can be collected into a distribution, and the standard deviation of that distribution can be calculated according to its type.

4. If there exists, or can be created, a twin security which a) is traded, and b) has the same risk characteristics as the project; the price S of the security will have the same volatility as V. Benaroch mentions two main cases where this is applicable: "when there is a publicly traded firm whose primary revenue generating services parallel the services that the target project would yield to generate payoffs", and "when the primary risk in the target project is due to reliance on a risky IT that is the main product sold by a traded firm" [16]. In this case, the volatility observed in S could be used for σ.

5. In cases that the sources of risk are separable, such as technical risk, competition risk, compliance risk, financial risk, etc., Benaroch proposes that overall volatility can be calculated from a combination of these risks. Where r_i is an applicable risk and $\sigma(r_i)$ is the contribution of r_i to σ, σ can be calculated using Equation 3.12. However, even if the risks are separable, determining the relationships between risks ($cov(r_i, r_j)$) is not a trivial problem in its own regard.

$$\sigma = \sum_{i,j=1}^{m} \left[\sigma(r_i) + cov(r_i, r_j)_{i \neq j}\right] \tag{3.12}$$

Ozkaya *et al.* additionally mention *implied volatility*, which is a method of calculating volatility for an asset that currently exists. The current price of the asset, along with the other standard inputs, can be entered into a pricing formula, which is then solved for σ [59]. This method, however, is not effective for predictions, as the project must already exist for this avenue to be available. Ozkaya *et al.* also mention that historical data from which to estimate volatility is frequently unavailable. Furthermore, they state that volatility can come from a number of disparate

sources such as "requirements changes, market response, or the changing price of an outsourced key design component". This supports the assertion of Bajracharya *et al.* and reinforces the notion that volatility is complex, complicated, and difficult to estimate in a general, all-encompassing, way. If neither past data nor enough information to perform a Monte Carlo simulation exist, then the simplest, and likely most accurate, course of action would be to have the stakeholders perform their own volatility analysis, taking into account all relevant sources of risk; this reported volatility could then be used with at least moderate confidence in value calculations.

3.5 Summary

In valuing a service, two approaches must be taken: the first in a general sense, determining the overall worth of a project to its owner; the second, placing a value on the flexibility inherent in the SOA style of development. The former of these can be performed though generic project valuation. To this end, I present three commonly used models: NPV, IRR, and MIRR. While each of these has its particular use, NPV is best suited to this research, and will form the foundation of the framework outlined in Chapter 5. To the latter, we associate project flexibility with real options, and present methods for calculating the value of these options: namely Black-Scholes, Black's approximation, and binomial lattices. While the Black-Scholes method is straightforward and easily computable in a closed form (with access to a standard normal CDF estimation function), it does not allow for American-style options, and modelling multiple interacting options is difficult at best. Black's approximation allows for American options, but still cannot handle multiple options. As such, the benefits of the the binomial lattice method—American option support, interacting options, and a visual representation of the option value time line—outweigh those of the other models for real option valuation. Regardless of the valuation model used, an estimate of project cash flow volatility is required; this is recognised as a hard problem, and numerous methods used to estimate volatility are reviewed.

Many inputs are common to numerous models; for ease of reference, I provide

a list of required variables, along with information on their use, in Table 3.1.

Variable	Definition	Model(s)	Comments
r_t, r, r_f	Risk-free rate or discount rate	NPV, IRR, MIRR, B-S, Black's approx., Binomial	r_t indicates the discount rate for time t, r indicates a constant rate for all t, r_f is $1 + r$.
I_t	Project income at time t	NPV, IRR, MIRR	
C_t	Project cost at time t	NPV, IRR	Can be predicted using cost estimation models from Chapter 2.
C'_t	Out of pocket project costs	MIRR	Only includes costs not financed by positive project income.
C''_t	Reinvested project costs	MIRR	Only includes costs that are financed by positive project income.
U	Spot price	B-S, Black's approximation	Current market price of asset underlying option.
S	Strike price	B-S, Black's approx., Binomial	Fixed price of option. Call = buy price; put = sell price.
T	Periods until option expiry	B-S, Black's approx., Binomial	"Periods" are counted in the same scale for which r is given.
σ	Volatility of underlying asset	B-S, Black's approx., Binomial	Provided in the same time scale as r and T.
Ξ	Set of expiries	Black's approx.	All possible expiries for American options.
V	Initial asset value	Binomial	Value of asset underlying option at $t = 0$.

Table 3.1: Input variables for value models.

Chapter 4

Related Work

Work related to this thesis can be divided into three main categories: software economics, architectural models, and software service valuation. Software economics, in this case, refers to aspects of software engineering where economic models contribute information to, and affect the outcome of, the decision-making process. Architectural models are similar, but on a larger scale; they attempt to model numerous aspects of the development process with a certain goal in mind, or from a certain viewpoint. They represent facts about the entire architecture of a system; in this context, these facts are economic, or utilize econometric theory as a mainstay of the model. The third category is much more specific, and only discusses work related directly to valuing software services, notably web services.

4.1 Software Economics

As software has become more flexible, economic factors, and value in particular, have come to play a larger role in development decisions; numerous papers have been published on the topic of software valuation under varying development strategies.

Favaro discusses the economics of reuse [33]. He presents and compares five valuation approaches: NPV, payback, average return on book value, IRR, and profitability index (PI). Payback refers to the number of times the software has to be used (or reused, as in Favaro's case) to recover the cost of investment. The general rule for decision-making using the payback method is to accept a project if the pay-

back can be obtained before a specified time. This leads to a number of concerns when using this model: cash flows are generally not discounted, the cutoff date before which the cost needs to be recovered is arbitrary, any cash flows after the cutoff date are ignored, and the model does not account for scale when comparing multiple projects of different sizes. Scale is important, and as such cannot be ignored; two projects could have the same predicted payback date of t, but the cash flows of one project could be much higher than the other, and would create considerably more profit than the other after t.

Average return on book value amortizes the value of the project over its lifetime, then takes the average of the value of the project, minus cumulative amortization, as the net average book value. This figure then divides the total cash flow for each period to produce an average return on book value percentage. If this percentage is greater than an arbitrary target percentage, then the project should be undertaken. However, this method does not account for differences in cash flow patterns: large inflows combined with large outflows are treated the same as small inflows with associated small outflows. Also, the method of calculating amortization for a project is not standardized, and is left up to the discretion of the particular accountant; this can lead to vastly different calculations of return on book value, limiting the comparability of such results across projects.

The profitability index (PI), or cost-benefit ratio, of a project is simply the present value of the project divided by the initial cost to create the project. The project should then be accepted if the PI is calculated to be greater than one. Like the payback technique, this ignores the the size of the project or the actual quantities of cash flows, undervaluing projects with larger flows. Additionally, the values calculated for two projects cannot be added together to determine the PI if both projects are undertaken.

NPV and IRR, as discussed in Section 3.1, are compared to the previous three approaches; Favaro reports that NPV is the superior technique for valuing reuse-based software development. He reaches this decision based on the fact that NPV can account for projects of varying scale, the NPV values of separate products are additive, and NPV allows for discounting of future cash flows.

Erdogmus studies the value of development strategies, including those based on the use of commercial-off-the-shelf (COTS) components [27, 32]. COTS development is generally flexible because of the ability to place external modules into the system, which can be upgraded or modified at a later time. Erdogmus presents two approaches for valuing COTS-based development: a comparison of COTS strategy against a custom development approach using NPV Incentive (NPVI), and a real option-based method. The first strategy simply compares the NPVs of the two development alternatives and distills this comparison down to a value: NPVI. This simplifies the decision of whether or not to undertake COTS-based development; if $NPVI > 0$, the COTS-based approach will be more profitable. If not, the developers should undertake a custom development strategy and produce their own software to suit their needs. The formula for NPVI is presented in Equation 4.1, where NPV_x if the NPV produced by approach x; C is the asset value (the total value of future predicted cash flows discounted from the time period in which they occur); M is the operation cost, or ongoing costs, discounted from the time they occur; I is the initial development cost.

$$NPVI = \frac{NPV_{COTS} - NPVcustom}{C_{custom} - M_{custom} + I_{custom}} \tag{4.1}$$

The second approach uses real options to model choices available during development, including the option to replace components (with other COTS products that better meet the needs of the application), the option to skip or delay upgrades of the underlying COTS components, the option to abandon the project for a termination value, and the option to migrate from a COTS-centric system to a custom design once the benefits of rapid development and early market penetration have been realized. This work uses the Black-Scholes formula to calculate the value of the options involved, with the option value added directly to the NPV value for each alternative.

Like Favaro, Erdogmus also reaches the conclusion that NPV is the most effective measure of a project's worth, however he mentions little of how to determine the values of the parameters in the formula. He also summarizes with a set of guidelines for assessing a COTS-centric development project, which include: con-

duct comparative evaluation between two development approaches, identify possible future states in which the project could find itself, account for flexibility, focus estimation on easily predictable variables, consider market conditions and business context (specifically when choosing a discount rate for NPV), and concentrate on what-if analysis instead of single-value data. These guidelines are not only specific to COTS-based projects; they equally apply to service-oriented development as well. This thesis follows these guidelines when developing a framework that can be used to compare the economic feasibility of a set of potential service-oriented projects.

4.2 Architectural Models

More recently, researchers have begun tackling the problem on a larger scale: creating frameworks and theories that operate on entire projects and model both costs and values in detail. Many of these incorporate a real option element for the valuation of flexible project aspects. Three works are most closely related to this thesis: that of Ozkaya, Kazman, and Klein, that of Bahsoon and Emmerich, and that of Baldwin and Clark.

Ozkaya *et al.* discuss the architecture of a system from a quality standpoint, embodying the notion that various quality attributes dictate the architecture of a project [59]. Quality attribute requirements can be met through the use of architectural patterns, and these patterns have inherent economic values that can be modeled as real options. This approach can be used to map future quality-based design decisions to economic terms. The report strives to determine how the option value can be extracted from architectural patterns, if this information can be useful for determining the optimal time to apply an architectural pattern, and if pattern valuation can be used to inform decision-making related to quality and business objectives. These issues are explored through the use of a sample case study based on a fictitious city information system (CIS) that contains an option to alter the modifiability and availability of the system through different architectural patterns. The binomial approach is used to value real options. Their results on the sample system sup-

port the hypothesis that real options are a viable method of performing value-based comparisons between possible architectural patterns. This work is closely related to the Architecture Tradeoff Analysis Method (ATAM) [45] and the Cost Benefit Analysis Method (CBAM) [8]: ATAM analyzes a system to determine whether architectural decisions lead to a satisfaction of desired quality attributes, while CBAM studies cost and benefit implications of architectural decisions, assisting developers in choosing which from a set of possible actions should be undertaken. Costs are elicited from the organization whose project is under analysis; benefits are given in terms of utility, where meeting quality attribute requirements will lead to an increase in utility.

Bahsoon and Emmerich develop ArchOptions, a model for determining the stability of software architectures, where *stability* refers to the ability of an architecture to handle changing requirements put forth by stakeholders or the environment [9]. A more stable architecture is generally beneficial as it can withstand requirement changes without necessitating a redesign of the system. This work quantifies the value of each possible future evolutionary change to the architecture by casting them as call options and using the Black-Scholes formula for valuation (later compared to the binomial valuation method in Bahsoon's PhD thesis [11]), with a greater value associated with a greater level of flexibility. Bahsoon and Emmerich apply this model to the field of refactoring by way of placing a value on the flexibility that a refactoring may introduce to a system [10]. The authors apply ArchOptions to a simulated case study from the literature, performing random refactorings to the code and measuring the expected maintenance savings of each refactoring. Each change to the code after the refactoring is performed is assumed to be an option; this allows the "option value" to be determined by finding the break-even point (the number of changes required to allow the maintenance savings to outweigh the cost of performing the refactoring). Any changes performed after the break-even point contribute to the option value. However, using the ArchOptions model the authors determine the break-even point to be ten changes subsequent to refactoring, yet Stroulia and Leitch, the authors of the paper on which the example is based, determine the value to be six [68]. There is no discussion of this discrep-

ancy, or how or why the ArchOptions model may be superior in this context.

Baldwin and Clark use a Design Structure Matrix (DSM) to map hierarchical or interdependent relationships between design elements in a project [13]. A DSM is a simple map denoting dependencies between modules, which can be analyzed using the Net Option Value (NOV) model to determine what architecture would create the most value for a system. The NOV model assumes that each module has an expected value normally distributed around zero and a variance that is directly correlated with the complexity of the module: to be more precise, variance increases linearly with the number of tasks the module can perform. However, as the module can be discarded if the value (X) is deemed to be below zero, the actual expected value only includes the positive outcomes, or $E(X^+) = \int_0^\infty X f(X)\, dX$, where $f(X)$ is the normal density function.

To generate options, which in this case are alternative architectures, a combination of six modular operators are employed to alter the system in various ways. These operators are splitting a system into sub-modules, substituting one module for another, augmenting the system with a new module, excluding a module from the system, inverting to create new design rules and standard components, and porting a module to another system. For each possible system, the NOV is the sum of the module values plus an overall system value for modules with additive value properties, or a combination of values if they are not additive. In this model, the architecture with the greatest NOV is chosen. Sullivan *et al.*, who treat software design as a value-maximizing investment activity, verify the potential usefulness of this approach for informing the software design process [69]. Baldwin and Clark, however, later revisit the issue and mention the difficulties in performing accurate valuation of complex systems, partially due to the lack of concrete datasets on which models can be tested [14].

4.3 Software Service Valuation

While most related to this research, few works have concentrated on the valuation of software services specifically. As web services are becoming such a pervasive

force in e-commerce, the lack of research in this area is surprising. This void may not be web service-specific though; Wiederhold discusses the deficit of software-related value research is his papers on software and web service valuation [76, 75], in which he stresses the need for such research since the value of a project is generally unrelated to the costs expended to create it. He concludes that the value of software is the income the project can generate, and that constant maintenance of a traditional software product can effectively transform it into a service, as customers will pay for updates to the software instead of new and separate products. Though he does not specifically mention valuing software flexibility, it can be construed as a portion of the income a project generates, albeit indirectly. Interestingly, Wiederhold also states that on average 5% of code is replaced in a project through normal maintenance activities, which increases "to 10% for code that is aggressively maintained" [76]. In this thesis, this value is used to determine default values for COCOMO II calibration parameters.

Also studying web services in particular, Van der Raadt *et al.* examine service valuation in the context of requirements engineering, developing the Business-oriented Approach Supporting web Services Idea Exploration (BASSIE) [73]. BASSIE combines a goal-oriented framework for determining alternative methods for reaching business goals with a value-oriented framework for estimating the economic worth of these alternatives. While the details and calculations of the underlying models are not provided, there is no mention of future flexibility; therefore, it is unlikely that real options constitute a part of the valuation process.

Perhaps the most prolific author in the area of web application economics, Emelia Mendes has performed numerous studies, particularly in the fields of cost and effort estimation, sizing metrics, and hypermedia applications[1]. While nominally similar, web hypermedia applications are created at a much higher level than the sourcecode-level applications studied in this work, with a concentration on content delivered though multimedia, JavaScript, HTML, and other web publishing media [56]. Instead of sizing metrics based on source code size and complexity, metrics include HTML page count, media file count, Java applet or JavaScript file

[1] See http://www.cs.auckland.ac.nz/ emilia/publications.html for an extensive list of publications

count, internal link complexity, and average media type counts. While Mendes's research is comprehensive and well published, it is dissimilar enough from this work as to not contribute towards its goals.

4.4 Summary

The economics of various aspects of software development have been studied in numerous contexts related to software services, including COTS development and generic reuse. These studies have analyzed a range of economic models for placing a value on the software project, including NPV, payback, average return on book value, IRR, and PI, as well as the custom model NPVI. NPV, and models inspired by NPV, are consistently chosen as the most effective approach to valuation based on its ease of application, ability to discount future cash flows to current terms, and final numeric value that is representative of an actual dollar amount, making it easy to compare projects of various sizes. Models have also been developed in an attempt to utilize architectural information, including those concerned with architectural quality and architectural stability. Software service valuation, or models applied to the architectures of services in particular, has not been extensively studied, particularly not in conjunction with a real option approach for valuing flexibility. The existing work on software services tends to be focused at a high level, not accounting for actual features of the codebase.

Chapter 5

Methodology

Building upon the work described previously, this section introduces and details the steps of the ROMAN framework. I use a running example in conjunction with an explanation of each step so the reader can see how the framework can be used in its intended environment. This section also discusses the ROMAN Decision Support System, or RomanDSS, which can be used to automate some steps of the framework application process. RomanDSS performs Monte-Carlo simulation when the input variables are not precisely known, generating a probability density for likely project value outcomes. In order to expedite the estimation process, RomanDSS also includes default values for COCOMO II maintenance parameters; Section 5.4 discusses how these default values are chosen.

5.1 Case Study Introduction

While describing the ROMAN framework I illustrate the discussed concepts with an example case study, which revolves around the operations of the service-oriented business MovieInfo, Inc. MovieInfo is a company that provides a showtime information web service, with a web-accessible interface, for theatres in many geographical regions. Due to the popularity of this service, the company believes that it will be able to increase its revenue by expanding its service offerings. To this end, it is considering two expansion possibilities: 1) create a web service that allows customers to purchase movie tickets online, or 2) create a movie information web service, complete with a recommendation system and the ability to purchase

selected movies. Both expansions could be performed with their existing hardware infrastructure. The company has determined that it can only secure the capital to undertake one of these expansions at the present time, and thus needs to decide which expansion to perform.

As I establish the ROMAN framework in Section 5.2, I will describe how each step applies to the MovieInfo example, and how the application of the framework will assist MovieInfo in choosing the correct path of expansion, accounting for the value of flexibility. As the framework applies to both possible expansion alternatives, for the sake of brevity I will only provide calculations for the second expansion option.

In terms of the information service, MovieInfo intends to generate income by placing advertisements in the movie information results; the company charges advertisers a cost per thousand impressions (CPM) of $5. Additionally, from the movie selling portion, they foresee profits of $15 per thousand impressions, on average, based on industry norms. Using this information, combined with that given throughout the example, MovieInfo will be able to determine which expansion alternative to undertake.

5.2 ROMAN Framework

Based on the McKinsey flexibility valuation process [55], the ROMAN framework involves five steps, each of which is designed to assist in the process of placing a value on a project, with the intention of helping an interested entity choose between alternative projects to undertake. This method of proper usage should be stressed: even though the framework will provide a numeric value in monetary terms as its final value estimate, this number should not be relied upon as being an accurate monetary prediction. Instead, the value should be compared to that produced from the analysis of other alternative projects, and the project with the greatest estimated value should be undertaken. This intended use guideline is given as a precaution, since more extensive real-world testing is needed to determine the absolute accuracy of the estimation results. This recommendation is in line with those from other

authors, such as Erdogmus and Vandergraff [32], who stress the need for what-if analysis over single-value data in economic valuation, allowing for the exploration of alternatives rather than relying on precise estimation figures.

The first step of the ROMAN framework involves performing traditional NPV calculations for each alternative. The NPV forms the basis of value calculations, but neglects the value of project flexibility, and will therefore understate the expected value of a project. The second stage models cash-flow uncertainty over the life of the project on an event lattice, allowing the valuator to determine the likely outcomes of the project. Steps three and four work to identify flexibility in the project: step three indicates commonly embedded options at the project level, and step four specifies how flexibility at the design level can contribute to project flexibility. Both of these steps are given as guidelines for areas to examine when searching for embedded options during project valuation. If applicable to the project under valuation, the value of the option can be calculated with the provided formulae. The final step in the framework combines the value of options located in steps three and four with the original event lattice, forming a decision lattice that indicates both the optimal exercise time for embedded options and the updated NPV of the project, including the value of flexibility. The conclusion of this step restates the traditional NPV rule in a modern context: choose the project with the greatest NPV, where NPV includes the value of flexibility afforded by the specifics of the projects in question. The five steps of the ROMAN framework are detailed below.

5.2.1 Step 1: Estimate Basic NPV of Alternatives

The first step in the ROMAN framework is to perform a traditional NPV calculation, devoid of any notion of the value of flexibility, for each alternative under consideration. The general formula for calculating NPV, discussed in Section 3.1.1, is given in Equation 3.1.

MovieInfo's information and retailing service is expected to receive 4 million requests per year, based on present service demand, yielding income of $80 000 annually, for the service's lifespan of 10 years. The initial cost of developing the service can be calculated with COCOMO II, the inputs for which are given in Ta-

ble 5.1. For descriptions of each input parameter, see Section 2.3. Note that Table 5.1 assumes Java as the programming language in which the project is being written, providing a 1/53 FP/SLOC ratio.

Variable	Definition	Value
Codebase	Size of existing codebase	200 FPs
DM	Percent design modified	40%
CM	Percent code modified	59.5%
IM	Percent retest impact	85%
UNFM	Progammer unfamiliarity	0.4
AA	Assessment and assimilation	0
FP/SLOC	FP/SLOC ratio (language-based)	1/53
Modified FPs	Original code modified (interchangeable with CM)	119 FPs
New FPs	Additional code required to achieve functionality	130 FPs

Table 5.1: Initial development COCOMO II input variables for MovieInfo movie information web service.

These inputs result in an initial cost of \$99 989.22, which I will round to \$100 000 for the sake of simplicity.

Each year, the service will incur costs of \$45 000 for maintenance of hardware and software, bandwidth fees, support, and shipping costs (as they will cover much of the cost of shipping movies). At an 8% cost of capital, this places the NPV for the service at $\sum_{t=1}^{10} \frac{80\,000-45\,000}{1.08^t} - 100\,000 = \$134\,853$.

5.2.2 Step 2: Model Project Uncertainty

With the basic NPV estimated, we create an event lattice to model future uncertainty on the cash flow [24]. To do this, we need an estimate of the project's cash flow volatility, σ; this input parameter is discussed in Section 3.4. Once estimated, volatility can be used to calculate the value multiplier of an increase or decrease in cash flow, which are u and d in Equation 5.1, respectively. T is the constant time per step multiplier for the lattice. The probability of an upward step on the lattice, p, is given in Equation 5.2, with R_k being the cost of capital for the project [55].

$$u = e^{\sigma\sqrt{T}} \quad d = \frac{1}{u} \tag{5.1}$$

$$p = \frac{([1 + R_k]^T - d)}{u - d} \tag{5.2}$$

$$CP_{t,j} = p^j (1-p)^{t-j} \binom{t}{j}, \quad 0 \leq t \leq N,\ 0 \leq j \leq t \tag{5.3}$$

The event lattice can then be created for N desired time steps as shown in Figure 5.1; each column is tT time steps in the future. The cumulative probability of being in any specific state in the lattice, CP, then becomes Equation 5.3, where j is the ascending row index (from the bottom).

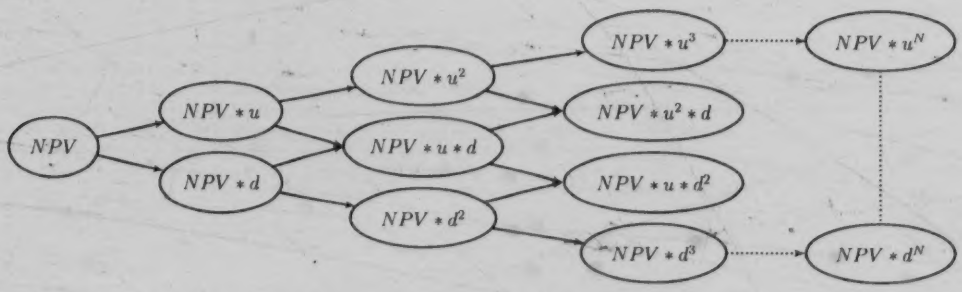

Figure 5.1: General event lattice.

The lattice for the MovieInfo example is shown in Fig. 5.2 for a volatility of $\sigma = 10\%$ and time steps of $T = 1$ year. Note that only the first 5 years are shown; as such, all example options are limited to a maximum expiry date of 5 years. Values indicate thousands of dollars.

5.2.3 Step 3: Identify General Project Flexibility

We now begin to look at the intangible aspects of value: those represented by real options. At this stage we examine general aspects of the service that lend themselves to the introduction of these options, and determine how these options affect the value of the project. Each option will directly alter the value of the project at the time that it is exercised; this change will propagate to $t = 0$ and modify the NPV for the overall project. As these options do provide value, one needs to be careful not to overlook this flexibility when valuating a project. Following is a list of common

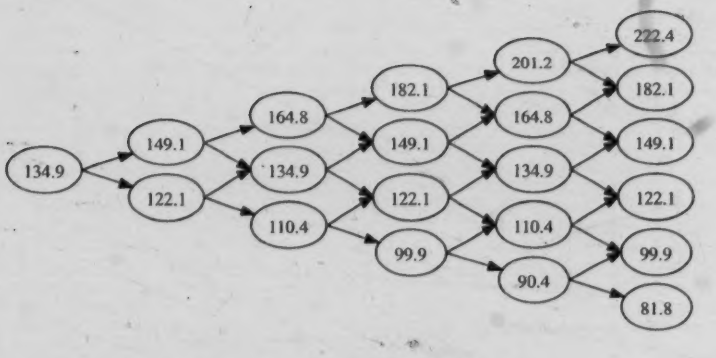

Figure 5.2: Event lattice for MovieInfo.

options that could apply to an overall service-oriented project; in addition to those listed here, other works have proposed varying categories of options and their affect on the value of a project [15, 31, 40, 55]. Note that these options are not exclusive; a project could embed multiple types of interacting options simultaneously.

Option to expand

This type of option can be used for a number of scenarios. Pilot projects embed options to expand; at any time, if the pilot is deemed successful, the service can be expanded into the full offering. Staged development is similar in this regard; upon completion of the initial stage, subsequent stages are optional, contingent on the success of the first stage. Likewise, deployed services can be expanded for a cost. The value of the node in which this option is exercised becomes $Ve - I$, where V is the previous value of the node, e is the expansion factor ($e > 1$), and I is the cost of expansion. This option is akin to a call option. Aside from flexibility at the project level, many expansion options exist due to the modular nature of services at the architectural level, as I discuss in Section 5.2.4.

Option to contract

The opposite of expansion options, contraction options allow for a service to reduce its costs with a reduction in the level of service provided. Services incurring a larger ratio of variable costs to fixed costs are more likely to contain options to contract. Take, for example, a web service that provides online retailers with the ability to

ship goods. Due to the Christmas season, fourth quarter demand for the service rises sharply, and first quarter demand drops sharply [74]. If costs are related to the number of simultaneous connections supported by the service, the ability to decrease the service level during the first quarter will reduce expenditures. To model an option to contract, the contracted node in the lattice becomes $Vc+I_c-I$, where c is the contraction factor ($c < 1$), I_c is the expenditure reduction due to contraction, and I is the cost to perform the contraction. Contraction options are akin to put options.

Option to defer investment

Occasionally, the present is not the optimal time for an investment. Delaying development of a service may have a number of benefits, including resolving uncertainty in the market value of the service, waiting for beneficial market conditions such as higher demand or reduced entry barriers such as legislation, or determining competitive ability. A product entering a new market may benefit from deferral until the owner can determine how the market will receive the product. When incorporating this option, the value at execution will become $NPV * e - V$, where e is the factor representing the additional income from deferral ($e > 1$), and V, the previous value of the node, is the opportunity cost of delaying the investment. If $(NPV * e - V) < 0$, then the option is not exercised (i.e., the service is not developed). This is different from Benaroch's definition of a defer option [15], in which the option is exercised when the deferral begins. In the definition used in this thesis, the option is exercised at the end of the deferral period; its cost is assessed as the income opportunity lost by not developing it sooner, and its value is estimated as the value of the service when developed. This definition is similar to holding financial options to mitigate financial risk, to be exercised only if conditions are sufficient for profit. This interpretation makes it similar to a call option. Note that e may be difficult to predict in some circumstances, such as in situations of uncertain customer demand; in others, such as deferring investment until legislation is conducive to market entry, this value can be fairly simple to estimate based on similar markets in unrestricted locations.

Option to abandon

Depending on the service, there may be value in abandoning it. This occurs if the owner is able to sell the service, or terminate it and liquidate its resources; a company that estimates the NPV of a service to be $20, but has been contacted by another company willing to purchase that service for $30, has the option to abandon for a profit. The development process of the service is one of the factors affecting its saleability; e.g., a service that is developed with stringent coding and documentation standards will produce a greater liquidation value than one with lax documentation and coding standards, based on the reuse value of the code. This option will affect the value lattice in two ways: it will remove all downward descendants of the exercised node, and the value of the exercised node will become $V_a - I_a$, where V_a is the liquidation or sale value, and I_a is the cost of discontinuing operation of the service. Like contraction options, abandonment options represent put options.

5.2.4 Step 4: Identify Modular Flexibility

General service flexibility is embedded in a given project based on market conditions, development practises, business models, and risk profiles. However, not all sources of flexibility are easy to identify at the project level; the fourth stage of ROMAN is to isolate flexibility value at the design level. The SOA tenet of loose coupling implies that systems are composed of collections of modules. Baldwin and Clark recognise the importance of modular design and establish a set of six operators that can be performed on modular systems; all modifications are a combination of these operators [13]. The adaptability that these operators provide adds value to the project, which can be expressed in terms of the options discussed in Section 5.2.3. In this section we discuss the inherent value of the six modular operators.

Splitting

Splitting describes separating a system into smaller component modules. This is important in the context of services, as large services can have the ability to be

decomposed into smaller, and possibly more profitable, services. This is the most basic case of splitting: dividing one stream of revenue into multiple. Effectively, the option to split a service in this way is an expansion option, as there will be an e increase in value, which is obtained by altering the software at a cost of I.

MovieInfo, at any time, could separate its recommendation system from the general movie information service and offer it as a separate service. They predict that this would lead to a 20% increase in usage ($e = 1.2$) between the two services, at a development and deployment cost of $I = \$30,000$. Figure 5.3 demonstrates the nodes where this option could be reasonably exercised; that is, any node where $Ve - I > V$.

Splitting also allows for common complex operations, which are discussed below.

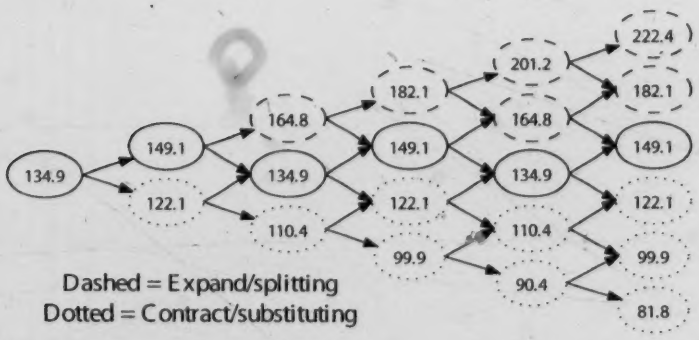

Figure 5.3: Profitable option exercise points.

Substituting

Substituting is the ability to replace a module in the system with another module providing the same functionality, but usually with a different level of service or cost. Generally, substitution is performed in one of two profit enhancing scenarios: in order to increase the revenue generated by the service, or to decrease excessive costs. The first scenario effectively represents an expansion option; the second, a contraction option. The first category encompasses actions such as upgrading a service to provide faster access or to handle more concurrent connections, either

through in-house development or by choosing a different provider for a third-party service. The second category performs the opposite function: decreasing the level of service when it becomes too costly to provide or is no longer desired by the customers.

MovieInfo currently has an embedded contraction option due to the ability to substitute one shipping company, used when customers purchase movies, for another. They currently ship with a reliable major courier, but could at any time modify their service to use the shipping functionality of the local postal service. The decrease in reliability and increase in shipping delays would devalue the service by an estimated 4%, but would save MovieInfo $5000 annually. The lattice nodes in which this substitution would benefit the company are shown in Figure 5.3.

Augmenting

Augmenting a service is adding another module to the system in order to raise the level of service provided to the customer. This could be performed as part of a differential pricing business strategy, with multiple services with increasing levels of functionality at increasing price points, or as a means to differentiate one's offerings from those of one's competitors [77]. It can also be used to increase the technical understanding of a service by introducing a logging module into the system. While not directly increasing the income generated by the system, this can reduce maintenance costs, which in turn increases value. Augmenting forms the main method for creating value-added services, and as such there are many possible motivations for augmentation. Common to all forms of augmentation, however, is the increase in the value of a service for a certain cost, or the ability to reduce costs while maintaining value, which implies that the ability to augment a system provides the owner with an expansion option.

Excluding

Exclusion is the modular opposite of augmentation, and refers to the removal of a module from a system. Removed modules can be those that are no longer desired by customers, consume more capital than they create, or are superfluous to

the operation of the service. The ability to exclude a module from a system is an instantiation of a contraction option where c is the reduction in service caused by the exclusion, I_c is saved cost of maintaining the module, and I is the cost involved in removing the module from the system.

Inverting

Inverting involves creating design rules. This is one of the more powerful operators in that it allows for standardisation, which can lead to the creation of standard, reusable, modules to perform common tasks across systems. At its most basic level, inversion leads to savings of maintenance costs. As design rules and standards are created, the software behind services becomes easier to understand, and therefore easier to maintain. Common solutions only need to be implemented once, reducing the size of the overall codebase. However, this does not represent an option; one does not generally plan to create a system and then extract presently visible commonalities at a future time. Inverting does lead to the presence of options when combined with other modular operators though, as is discussed below.

Porting

Porting is similar to inverting in that it allows a current module to perform the same task in another system. However, an inverted module does not change the design of the system. Porting, on the other hand, requires development other than the ported module; a ported module also requires the development of translator code to work effectively in the new system. The need for interface code often occurs with third-party, or COTS, components [32], and increases the costs associated with porting. As porting allows for the use of components that do not fit precisely into the modular framework, a wider range of solutions to a given implementation problem are possible. In this way, porting can be used for augmenting or substituting modules in the service.

Following the running example, MovieInfo has the option of substituting another back-end movie information data source, such as that of IMDb.com[1], into

[1] ©1990-2008 Internet Movie Database Inc.

their service. This would increase the quality of their search results, leading to greater service usage and more orders, but would require MovieInfo to create software to handle IMDb's database structure. MovieInfo would also have to pay content licensing fees. This option would, like the first category of substitution operators, be representative of an expansion option.

Complex Operations

Complex operations are composed of two or more component operators and performed as an atomic unit. As such, the net value profile is different from that of a simple combination of the underlying operators.

Split-exclude This operation is performed when part of a module, as opposed to an entire module, is to be excluded. The value of this will be identical to that of a simple exclusion, except the cost will be greater, as more development effort will be required to first split the system, and then remove the desired module. When analysing an architecture for options, options based on this operation should be treated as exclusions.

Split-substitute Similar to split-exclude, this operation is required for substituting part of an existing module. During option analysis, split-substitute operations should be treated the same as substitution operations, but with the added cost of splitting the system incorporated into the value calculation.

Split-invert As with the previous two operators, this is performed to extract a subset of a module before conversion to a common module. This is similar to inversion in that no options exist based solely on this operation. It can, however, form the basis for the next two complex operations.

Invert-augment Can also be split-invert-augment. These operations are used to create service frameworks to facilitate future development. MovieInfo, at some point in the future, intends to expand into the soundtrack-selling business as well as movies. With their modular design they can create a generic product-selling framework through inversion, use the framework as the basis of their movie-selling service, and then augment the system with a new soundtrack-selling module. This development would represent an expansion option: the cost would be the develop-

ment cost of inversion and creating the new service, and the value would be the increase in revenue as a percentage of total sales.

Invert-exclude Can also be split-invert-exclude. These complex exclusions can be used to perform service differentiation, similar to augmentation. Unlike augmentation though, this is not a unary operator; the core functionality must be inverted before differentiation can be performed. The result of the inversion is then used as the base for all levels of service, and various other modules are excluded from the original service to reduce functionality and price. Counterintuitively, the ability to perform this operation in the future is an expansion option: the product line is being expanded, even though a service is being contracted. The original service remains, but new services are introduced at lower price points. MovieInfo has the option of performing this operation by offering a reduced functionality service—one without the recommendation system—with fewer advertisements displayed. The reduction in ads would make the service more attractive to some people who would otherwise not use either service, but this service would also cannibalise income from the original service. This results in many more sources of cost and value than inversion or exclusion alone.

5.2.5 Step 5: Calculate NPV Including Flexibility

With all sources of flexibility determined, the final step of ROMAN is to model this flexibility in a decision lattice and calculate the final NPV of the project. The decision lattice is formed from the event lattice combined with all relevant option values for the project. To illustrate this point, consider the expansion and contraction options described in Section 5.2.4. To demonstrate overlapping options, also consider an abandon option with a sale value of \$110 000 and no cost. The beneficial exercise times of these options are shown in Figure 5.4.

The decision lattice can now be used to find the optimal exercise time for each option, and to calculate the NPV of the service that embeds the options. Starting at the end of the lattice ($t = 5$), we determine which has the maximum value: any applicable options (one each of expansion, contraction, and abandonment options in this case), or the current node value (i.e., no option). We then propagate the value

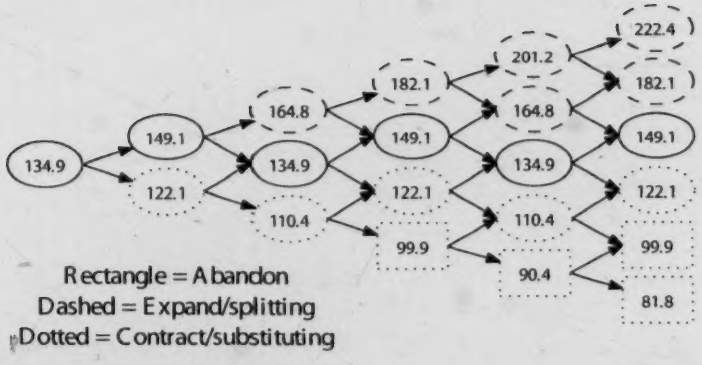

Figure 5.4: Profitable option exercise points including overlapping options.

back, at each stage checking the possible option values against the propagated value. If at any stage an option value is greater than the propagation value, the option is exercised at that point and the value produced by the option is then propagated back. To perform the propagation, we use the portfolio replication technique, which models the value of options as a combination of B risk-free assets and a portfolio of N assets with identical risk and value profiles as the project being valuated [55]. The propagation value $P_{t,j}$ of node (t, j), where t is the column index and j is the ascending row index, is shown in Equation 5.4. In this equation $V_{t,j}$ is the original node value as seen in Figure 5.3, $O_{t,j}$ is the post-calculation node value (which is the max of $P_{t,j}$ and the various applicable option values), and r_f is the risk-free rate, such as that guaranteed by government bonds (note that r_f is equal to $1 + r$ from Section 3.1.1).

$$P_{t,j} = V_{t,j} \times N + B \tag{5.4}$$

$$N = (O_{t+1,j+1} - O_{t+1,j}) \,/\, (V_{t+1,j+1} - V_{t+1,j})$$

$$B = (O_{t+1,j} - V_{t+1,j} \times N)/r_f$$

The final decision lattice for MovieInfo is shown in Figure 5.5, with the locations of option exercise indicated. The NPV for the service including options is shown in the root node; for this example, the final NPV is \$139.4k, leading to a total combined option value of \$4491.

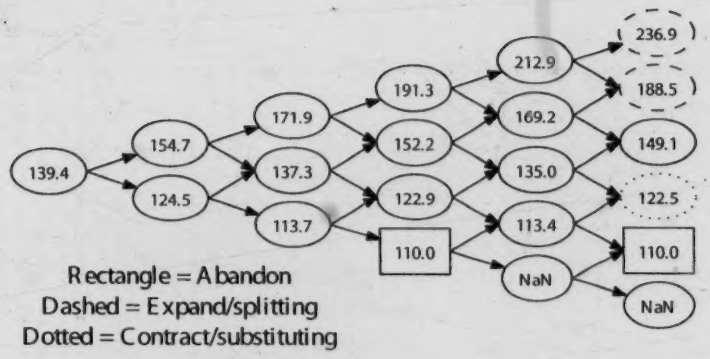

Figure 5.5: Final decision lattice with optimal option exercise times.

With the flexible NPV of each alternative project estimated, the project with the greatest NPV can be chosen as the one to undertake, given that the NPV is greater than zero. Clearly, if resources allow, more than one project can be chosen for implementation in order to maximize profit. In the case of MovieInfo, we have already established that they only have available resources for a single expansion. Therefore, if the service to offer the movie tickets online has an NPV greater than \$139.4k then it should be undertaken; otherwise, MovieInfo should proceed with the information and movie sales service.

5.2.6 Alternative Calculation Procedure for Step 5 - Risk Neutral Valuation

Step 5 of the ROMAN framework utilises the portfolio replication technique for node value propagation. There is an alternative method to perform this task that can be used as well: risk neutral valuation [55]. This method adjusts the probabilities of entering a successive state to account for risk. These probabilities are used, in conjunction with $V_{t+1,j+1}$ and $V_{t+1,j}$, the original up and down node values at $t+1$, and discounted by the risk-free rate, to determine $O_{t,j}$, the new value for the node at (t, j). The formula for calculating the risk-adjusted probability of an upwards movement at any given node, $p^*_{t,j}$, is given in Equation 5.5.

$$p^*_{t,j} = \frac{r_f - d_{t,j}}{u_{t,j} - d_{t,j}} \tag{5.5}$$

When using risk neutral valuation u and d are not calculated as in Equation 5.1, but are now directly based on $V_{t+1,j+1}$ and $V_{t+1,j}$. The risk neutral versions of $u_{t,j}$ and $d_{t,j}$ are given in Equation 5.6.

$$u_{t,j} = \frac{V_{t+1,j+1}}{V_{t,j}} \quad d_{t,j} = \frac{V_{t+1,j}}{V_{t,j}} \tag{5.6}$$

$P_{t,j}$, the propagated value of $V_{t+1,j+1}$ and $V_{t+1,j}$ as in Step 5, then becomes a probability-weighted combination of $V_{t+1,j+1}$ and $V_{t+1,j}$, discounted by the risk-free rate, as is shown in Equation 5.7.

$$P_{t,j} = \frac{p^*_{t,j} \times V_{t+1,j+1} + (1 - p^*_{t,j}) \times V_{t+1,j}}{r_f} \tag{5.7}$$

This $P_{t,j}$ can then be used in the same way as in Step 5: $O_{t,j}$, the new value for node (t, j), will then become the max of $P_{t,j}$ and the various applicable option values that can be exercised at that point.

The main difference between the risk-neutral valuation method and the portfolio replication technique is the incorporation of risk information into the probability of upward movement; some valuators may find this information more useful than the original scenario probabilities. Both approaches will yield identical results for $O_{0,0}$, the NPV of the system after all calculations have been performed.

5.3 RomanDSS - The ROMAN Framework Decision Support System

In order to facilitate the calculation of project costs and values, I developed RomanDSS, a decision support system designed to simplify and study this process. The application uses process descriptions provided by BPEL (Business Process Execution Language) files to model the prospective service under consideration. BPEL was selected for five main reasons: 1) it is being increasingly adopted by organisations, as an increasing numbers of tools supporting it have become available; 2) in addition to supporting the specification and orchestration of web-service compositions, it is currently evolving to represent interactions with humans involved in the orchestration process (a necessary step to meet the requirements of

a business-process modelling language); 3) its open XML representation format makes its parsing and analysis straightforward; 4) the structure of the orchestration expressly models modularity, with modules being accessed through service calls; and 5) the BPEL process can easily be created at the architectural stage, before the system is implemented.

BPEL specifies how services are delivered as compositions of other coordinated services. The composed services can be simple, i.e., implemented in a "native" programming language, or composite, i.e., implemented as further BPEL compositions. Costs and values for the web service under evaluation are determined through analysis of the BPEL compositions and the auxiliary web services that it accesses. For example, the MovieInfo service uses an information retrieval service and a recommendation service; those two auxiliary services would be treated as modules under the ROMAN framework. If the auxiliary service is itself implemented as a BPEL file, the DSS recursively analyses it as a module composed of sub-modules.

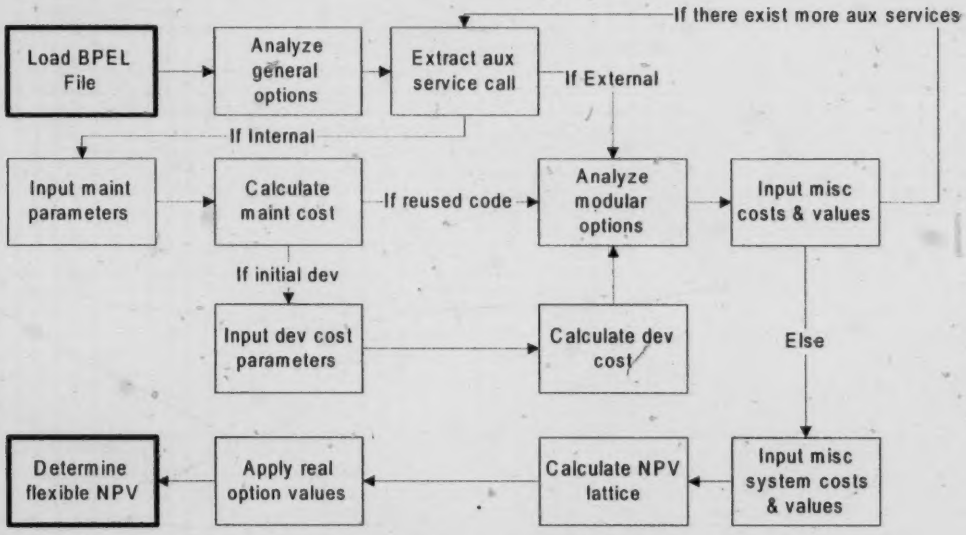

Figure 5.6: RomanDSS execution graph.

The operation of the tool, as is shown in Figure 5.6, is as follows. The user initially loads a BPEL file into the system and performs real option analysis on the overall project by entering applicable options as per Section 5.2.3. Any options de-

termined to exist at this stage are attached to the project as a whole. The DSS then performs modularity analysis on each service call extracted from the BPEL file. If the user tags a call as "internal" to the system, i.e., controlled by the same entity as the BPEL-based process and is in charge of maintenance or modifications, then the user is prompted for typical maintenance values, such as the average percentage of the codebase that is modified during maintenance. If the module is not yet developed, they are also prompted for initial development cost parameters. COCOMO II is then used to estimate the actual cost of these activities. At the next stage, the DSS prompts the user to consider modular operators and how they can create real options for the service, as discussed in Section 5.2.4. If any options exists at this stage, they can be attached to the module from which they derive their value. If there are any miscellaneous costs or values associated with the module, they are also entered at this point. Once all modules have been analysed, the user is prompted for miscellaneous costs and values that apply to the overall system, such as the actual income from customer usage. The DSS then performs basic NPV calculations, creates an event lattice for the project, determines and propagates option values, and reports the final flexible NPV of the system.

Note that any and all inputs to the application, be they cost or value related, can be in the form of a probability distribution. Currently the application supports the following distributions: uniform, Gaussian, upper- and lower-bounded Gaussian, beta, chi-square, exponential, and gamma distributions, as well as a simple unweighted list of possible values. If any inputs to the system are stochastically distributed, instead of performing a simple calculation of the values, the DSS performs Monte-Carlo simulation by repeatedly sampling from the input distributions to form a resulting output distribution. The user can choose the resolution of the output distribution, where higher resolutions imply more sample iterations and a more time-consuming process. Once the DSS has completed the sampling process, statistics such as mean, variance, maximum encountered value, and minimum encountered value are displayed along with a graph of the resulting distribution.

5.4 Default COCOMO II Maintenance Parameters

In an effort to simplify the set of inputs required to perform a ROMAN analysis, I have endeavoured to determine default values for maintenance parameters required for estimation. While obviously the estimation would be more accurate with parameters tuned to the specific project in question, these parameters can be used for times when a rough estimation is needed quickly. The parameters I examine are CM, DM, and IM, the percentage of code modified, the percentage of design modified, and the retest impact of a change, respectively. As indicated in Section 2.3.1, Leitch defines design modification as the percentage of methods changed during maintenance [49]; I adopt this definition for DM. As a basis for parameter analysis I also accept Wiederhold's statement that 5% of a project's source code is replaced during maintenance per year, or 10% under an agressive maintenance routine.

To determine average parameters I collected a set of 34 open source projects written in Java, taken from the Internet category of software projects on Sourceforge.net[2]. These projects and associated vital statistics are shown in Table 5.2. Coverage is presented at the method level, and is the percentage of the methods in an application that directly or indirectly depend on the other methods in the application. For example, an application with methods A, B, and C would have a coverage of 66.7% if A depended on B and B depended on C (which would create a dependency from A to C due to transitivity). Each method also depends on itself. Out of the nine possible dependencies, six exist: A depends on A, B, and C; B depends on B and C; C depends on C. Coverage is used to measure the impact that a change will have on a system, as any method that depends on the changed one will have to be retested (in optimal test conditions). To gather SLOC count information I use LOCC v4 [25], modified to count logical SLOCs according to the rules outlined in the COCOMO II.2000 model definition manual [19]. For data and control dependency information between methods I use Dependency Finder v1.1.1 [71], modified to better account for data dependencies.

I performed two sets of experiments on this set of applications: one intended to

[2] ©2008 SourceForge, Inc.

Project	SLOCs	Packages	Classes	Methods	Coverage %
52north-WNS-v2-00-00	835	1	6	44	15.42
Azureus_3.0.1.6	179919	393	2066	18772	1.02
FCKeditor-2.3	599	4	5	83	4.22
KeePassJ2ME-1_0_3	2374	14	41	289	2.82
aTunes_1.6.5	1626	9	32	281	1.77
ajaxtags-1.3-beta-rc4	2092	3	27	455	1.50
contineo-2_5_2	9144	35	213	1046	0.69
dconfig-lib-1.0	2145	7	43	328	7.03
dconfig-ui-1.0	3020	4	47	233	1.44
freemarker-2.3.10	26278	20	487	3036	7.79
freemind-0_8_0	15493	14	279	2269	3.22
grinder-3.0-beta33	22449	33	468	2626	1.55
hibernate-3.2.4.sp1	56952	75	994	10908	0.51
hsqldb-1_8_0_7	42700	11	266	4416	17.61
htmlparser1_6_20060610	21177	27	230	2441	2.75
htmlunit-1.11	9679	10	219	2110	1.57
itext-2.0.4	58007	28	419	5843	4.99
jbpm-identity	786	7	19	151	3.78
jbpm-jpdl-3.2.1	16482	62	344	2787	2.77
jena-2.5	19092	22	394	3373	0.44
jetty-6.1.4	21613	19	216	2703	1.06
jin-2.14.1	25438	47	348	3370	1.39
jodconverter-2.2.0	471	3	12	72	5.48
jodconverter-cli-2.2.0	53	1	1	3	44.44
jpivot-1.7.0	5801	9	84	805	1.16
jwebunit-1.4.1	1108	5	19	298	2.96
openxava-2.2	21744	48	347	3790	3.98
phex_3.2.0.102	2057	4	37	290	1.29
salamandra-web-0.1a	1409	15	61	302	1.02
servingxml-0.7.1a	22798	53	878	4168	0.13
soapui-1.7.5-beta2	1337	4	27	164	11.67
stendhal-0.61	2765	4	58	388	1.51
taste-1.5.5	53	2	2	16	8.98
tvbrowser-2.5.2	6586	14	88	929	1.25

Table 5.2: Projects and their statistics used in the COCOMO II default maintenance parameter study.

determine corresponding values of DM and IM for a given CM, and one to determine CM and IM for a given DM. The DM and IM results for CM=5% are used as the default maintenance values in RomanDSS; however, I explored DM and IM values for the set of CM={1,3,5,8,10,15,20,25,30,40,50}, and CM and IM values for DM={1,3,5,8,10,15,20,25,30,40,50} as well for completeness. Small projects tend to have parameter values that do not follow the application norm, and as such I ignore any application with fewer than 1000 logical lines of code. This reduces the experimental set from 34 applications to 28. See the discussion at the end of Section 5.4.2 for notes on usage guidelines for the formulae developed herein.

5.4.1 Experiments with a Controlled CM

For each test with a constant CM, the given percentage of code needs to be "modified". This is done in a simulated manner, with virtual lines of code randomly selected from a weighted list of methods. Each method is assigned a weight based on the SLOC count for that method, where the weight is the overall percentage of lines of code for the entire application that fall into that method. This makes the chance of any particular line of code being chosen as relatively equal. Note that the actual lines of code are not seen by the experimental apparatus, just methods and the SLOC counts contained therein, and as such each selected line of code merely increments a counter associated with a particular method. If a randomly selected method's counter is equal to the number of SLOCs in that method, the process is repeated with a different selection. This random selection process is repeated until the number of lines of code is equal to the desired CM percentage.

To determine the DM value for the selection, the number of methods with counters greater than zero are summed, and this number is divided by the total number of methods in the application. IM is determined though creating a map of all of the methods that depend on the selected methods (those with counters greater than zero), either directly or indirectly, and dividing that number by the total number of methods. For each application in the experimental set this process is repeated 100 times and the average result is taken per application. The combined results of this study over all sample projects are presented in Table 5.3. Figure 5.7 relates CM to

IM and DM.

CM	Avg DM	DM Stdev	Avg IM	IM Stdev
1	6.360749415	1.551097932	48.83809785	13.92333818
3	16.01217663	3.236440421	65.89568627	11.26959298
5	23.51940419	4.139720274	73.59384845	9.532340645
8	32.29893554	4.921004602	79.88952632	7.867964247
10	37.12470759	5.176156082	82.58958388	7.076503737
15	46.79292867	5.398315798	86.95845365	5.694858228
20	54.21561723	5.358990716	89.65802422	4.656470882
25	60.10419981	5.032901364	91.51219064	3.933940606
30	65.06603158	4.820944778	92.88688493	3.384294506
40	72.81675984	4.110380078	94.8099028	2.491083965
50	78.60740136	3.28797926	96.084518	1.878408751

Table 5.3: Experimental results of CM tests.

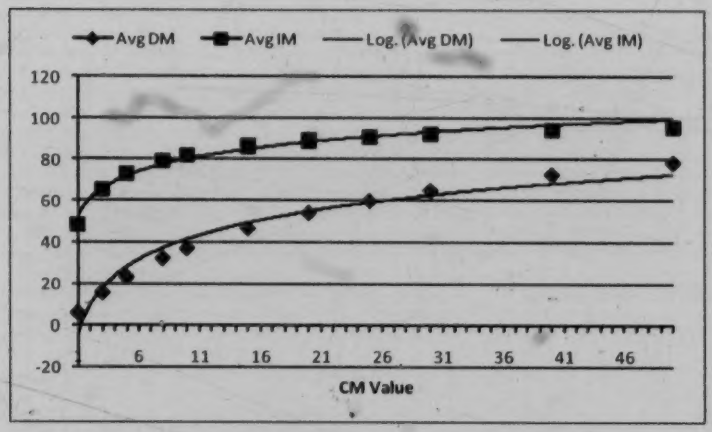

Figure 5.7: IM and DM values for given values of CM.

The logarithmic nature of these results leads to Equation 5.8 for estimating an average IM value from a given CM of randomly selected code, which has a coefficient of determination (R^2) of 0.976, and Equation 5.9 for DM, which has an R^2 of 0.960. Naturally, CM must be greater than zero in these formulae.

$$IM = 11.97ln(CM) + 52.65 \tag{5.8}$$

$$DM = 19.51ln(CM) - 3.104 \quad (5.9)$$

5.4.2 Experiments with a Controlled DM

I undertake a similar experiment with a constant DM in order to determine a corresponding value for CM and IM. This is performed by randomly selecting methods until the desired percentage of methods has been obtained. With this list of methods, sum the SLOC count for each method and divide this value by the total SLOC count for the application to obtain a value for CM. Determining IM is done in the same way as with the CM experiments: create a list of all the methods that depend on the selected methods, and divide the cardinality of that set by the total number of methods. Again, this process is repeated with each application 100 times and averages are taken across the entire set of applications. The results of these experiments is shown in Table 5.4 and in the relational graph in Figure 5.8.

DM	Avg CM	CM Stdev	Avg IM	IM Stdev
1	3.832716607	1.7017775	24.3326265	14.47211476
3	10.1469635	2.933153305	37.56468931	13.5861479
5	15.5584664	3.818518868	45.08329871	12.65929859
8	22.99474106	4.773549104	53.55044246	11.38571487
10	27.51159472	5.278015573	57.58744717	10.53378214
15	37.28791439	6.112093846	65.36621922	9.202003927
20	45.57440559	6.422309984	70.98016092	7.904978058
25	52.70482176	6.344153587	75.3477145	6.98743319
30	58.90850723	6.185517718	78.81473336	6.170869353
40	69.18450124	5.442527016	84.33309662	4.809262674
50	77.26781461	4.541087096	88.43235722	3.73653719

Table 5.4: Experimental results of DM tests.

From this information we can develop general equations for IM and CM based on the value of DM; these formulae are given in Equation 5.10 and Equation 5.11, respectively, with R^2 values of 0.990 and 0.993, where DM must be greater than zero. Equation 5.10 models a logarithmic sequence, while Equation 5.11 is a power trend[3]. The reader will note that values produced from these equations are not in-

[3]While divergent at high values of CM, the best-fit logarithmic approximation provided a signif-

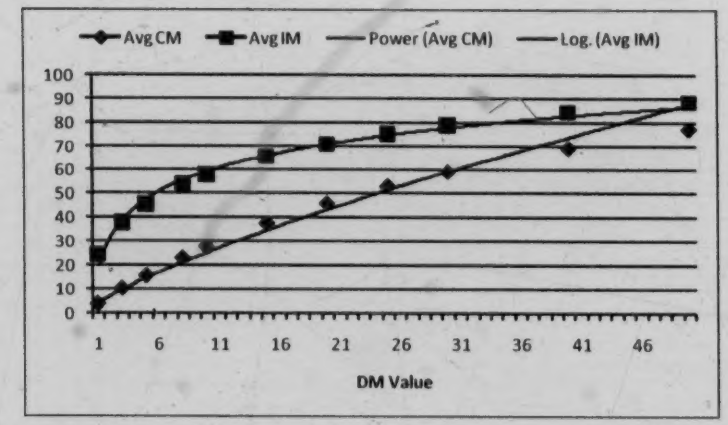

Figure 5.8: IM and CM values for given values of DM.

terchangeable: A DM calculated from Equation 5.9 cannot be then entered into Equation 5.11 to produce the original CM value. These sets of equations are meant to be used in different circumstances based on the style of change to be made to the project code. If seemingly random and independent lines of code are to be changed, as is common with simple bug fixes and many small maintenance activities, the formulae in Section 5.4.1 should be used. If the maintenance is more targeted to specific methods or classes, such as rewrites of sections of code, then the formulae in this section should be used. In this way the maintenance activities mirror the tests performed: in the first method maintenance is performed on a per-SLOC basis, whereas in the second method it is performed on a per-method (or larger) basis. Intuitively, this division is correct: many one-line changes will affect a larger percentage of methods than SLOCS, but changes that change entire methods or classes, and will occur with a greater probability in the methods containing the most SLOCs, will alter more SLOCs than methods.

$$IM = 16.98ln(DM) + 20.23 \tag{5.10}$$

$$CM = 4.331(DM)^{0.769} \tag{5.11}$$

icant decrease in accuracy, with $R^2 = 0.907$.

5.5 Summary

In this section I establish the ROMAN framework, which is composed of five main stages. These stages, when applied to a set of alternative projects to undertake, will allow a decision maker to choose the most valuable development direction. The first step in the framework is to estimate the basic NPV of the various possible projects. This NPV does not include the value of flexibility, but will provide a foundation for the complex valuation process. The second stage creates a model of the uncertainty attributable to a project using a decision lattice. This lattice shows the expected outcomes of positive or negative influences on the value of the project over time. Flexibility is introduced in step three, where the valuator is prompted to consider sources of flexibility, such as the ability to expand, contract, abandon, or delay investment in a project. These are modelled as real options. Step four also deals with flexibility translated to real options, but from a lower-level perspective. Flexibility exists through the ability to alter a modular project, such as a service, through the use of Baldwin and Clark's six modular operators: splitting, substituting, augmenting, excluding, inverting, and porting. These operators allow for options to be created for each module and the project as a whole. The final stage of ROMAN, step five, places a value on the collection of options that has been selected for use in the project. Combined with the original NPV value, this stage will produce a value for the project that includes flexibility; this value can then be compared with those of the other alternative projects, and those with the greatest values can be chosen for development.

I also develop RomanDSS, the decision support system, which implements the calculations involved in the ROMAN framework and assists an interested party in the decision making process. RomanDSS also allows for probabilistic inputs, which causes the system to perform Monte Carlo simulation to produce an expected value for the project, as well as variance and other distribution statistics.

In an effort to expedite estimation, though at the cost of accuracy, I also investigate default maintenance parameters that can be used in the COCOMO II estimation aspect of the framework. Based on previous work stating an average of $CM = 5\%$

per year I determined default values of $DM = 23.5\%$ and $IM = 73.6\%$. Aggressive maintenance, which would lead to $CM = 10\%$, results in defaults of $DM = 37.1\%$ and $IM = 82.6\%$. I also experiment with a range of CM values to determine their corresponding DM and IM counterparts for SLOC-targeted maintenance, and a range of DM values to determine their related CM and IM values for method- or class-targeted maintenance. Through these experiments I am able to determine a general formula for these inputs given the type of maintenance being undertaken.

Chapter 6

Conclusions

6.1 Summary

This work examines the problem of estimating the comparative expected value of alternative service-oriented applications, such as web services, based on the flexibility present in modular systems. Chapter 1 presents the problem, including motivation as to why it is important. It also gives a brief overview of the methodology I take to approach the problem. Chapter 2 provides a discussion of models for estimating software development cost, as well as the details of COCOMO II.2000, which I employ in this work. Chapter 3 handles the other half of the valuation problem: that of estimating net value for a project. This includes an introduction to real option theory—modelling available real-world choices after financial options to calculate their value—that I use in this work to place a value on flexibility afforded by modular service design. Chapter 4 discusses previous work in the area of software project valuation and presents other studies that utilise real option theory for valuing flexibility.

In Chapter 5 I develop the ROMAN framework, which is a structured valuation framework for estimating the net value of a project. ROMAN performs a traditional NPV estimation on the potential project, taking into account the volatility of expected cash flows, and then compliments this information with the value of flexibility. This is done by identifying areas of flexibility in the overall project, which is then combined with areas of flexibility created by the modular design specific to the system. To standardise the process of analysing modular flexibility, I incorporate

Baldwin and Clark's set of modular operators, which can be combined to produce any change to a system. Because of this attribute, these operators can be analysed to determine all areas of possible change to the system as well. The expected values for projects estimated by this framework can be compared to one another to assist in the choice of which project to develop from a given set of projects; this action will help maximize the return on investment of service development.

My current work in this area involves creating a decision support system, RomanDSS, to assist individuals who desire to use this framework for project valuation, which I discuss in Section 5.3. Using the DSS for the framework evaluation produces numerous benefits, including the ability to easily model 'what-if' scenarios, where a subset of the parameters are improved to the detriment of others. It also allows for inputs to be probability distributions, instead of simple averages. In this case, Monte Carlo simulation is used to determine the resulting distribution of the expected project value. This allows for uncertain input information to be modelled more accurately, and as a result the variance of future cash flows can be estimated. Another benefit of using the DSS is automated development effort calculations, performed by incorporating the COCOMO II effort estimation model into the system. This can be used to predict both development and maintenance effort, reported as the time required to perform a change, based on the quantity of requisite changes to the codebase. As gathering input parameters for a COCOMO II calculation can be a time consuming process, in Section 5.4 I perform a study to determine defaults for three main inputs associated with the maintenance calculation: the percentage of code modified (CM), the percentage of design modified (DM), and the retest impact of a set of changes (IM). Under normal maintenance conditions these values are $CM = 5\%$, $DM = 23.5\%$, and $IM = 73.6\%$. In an agressive mainenance environment these defaults increase to $CM = 10\%$, $DM = 37.1\%$, and $IM = 82.6\%$.

Overall, the main contributions of this thesis are threefold:

1. The first and foremost contribution is the development of the ROMAN project valuation framework, which includes associating modular design operators with commonly accepted real options for valuation purposes. The framework

presents valuation techniques in a structured approach, allowing for uniformity of project assessment for comparison purposes between alternative proposed projects. This allows interested parties to choose which from a set of possible services to develop by determining the project(s) that will provide the greatest NPV, including indirect value created from areas of flexibility within the project.

2. Secondly, I introduce RomanDSS, a decision support system for performing the steps involved in the framework, thus reducing the amount of calculation effort required on the part of the valuator. To accommodate uncertainty in the estimation environment, RomanDSS also allows for probabilistic inputs. RomanDSS handles these inputs by performing Monte Carlo simulation, producing a probability distribution for the resulting net value estimation. This flexibility allows the ROMAN framework to be used in situations with inexact parameters, providing an earlier opportunity for determining rough value estimates.

3. Thirdly, in a further effort to expedite the estimation process, I study default values for maintenance input parameters in the COCOMO II model. While the primary use of these parameters is their incorporation into RomanDSS, I also developed formulae for calculating the maintenance inputs for two contrasting types of maintenance activities. Having access to default parameters for maintenance cost estimations allows the valuation process to be applied in a more rapid manner, permitting more timely decisions regarding the development of a service.

6.2 Future Work

In the long term, future work needs to be performed to verify the value of real option valuation on software projects through real-world studies. The reason that I promote the comparison of projects through the framework instead of individual service valuation is that real option theory has not been extensively tested on software projects. Until this changes, I believe that estimated values can be used for

decision purposes, but not taken as precise expected values for individual projects. This situation needs to be rectified through studies of the real option value related to real software projects: those developed with the goal of producing profit, not merely in an academic setting. This will likely not be a simple task; it has been my finding that the data required to perform large real-world experiments is not generally publicly available.

I have adopted COCOMO II as the software-development cost model because it is designed to account for evolutionary software development. However, it is clear that the model will have to be extended in order to represent the particular idiosyncrasies of the service-oriented development paradigm. As it is, COCOMO II estimates the cost of adapting source code; however, SOA development also involves developing and adapting declarative composition specifications, which may be a fundamentally different process. Specifically, research into the cost of creating a service composition is required. Other cost models, including those based on COCOMO, need to be further examined for usability. The main impediment to this area of future work is the immaturity of cost models that would be more suited to this research, such as COSOSIMO and CORADMO; as applicable models are developed and matured, work needs to be performed to compare their usefulness in this type of research against COCOMO II.

On the topic of COCOMO II, I would also like to repeat the default maintenance parameter study with a much larger dataset in order to verify the values determined herein. While the resulting values lend themselves to a formulaic representation with quite high accuracy, a test repeated with more data may yield different results. Experiments to verify Wiederhold's CM declarations should also be performed to ensure precision.

The final area of future work relates to RomanDSS, as certain improvements would increase its usability and ostensible net value. The work flow needs to be optimised and made more user-friendly, more assistance should be given to the user when selecting real options, and experiments to determine the sensitivity of the final value estimate to various input parameters should be performed in order to gain the most information from the process. RomanDSS would also be more

useful if it contained more extensive support for modelling architectural operators as real options, allowing for a more concrete alignment of business process changes to an option representation. However, as with many software research projects, improvements can be made *ad nauseam* with little return on invested effort, even accounting for the value of future flexibility.

Bibliography

[1] Chris Abts, Barry Boehm, and Elizabeth Bailey Clark. Cocots: A cots software integration lifecycle cost model - model overview and preliminary data collection findings. Technical Report USC-CSE-2000-501, Center for Software Engineering, University of Southern California, March 2000.

[2] United States Central Intelligence Agency. *The World Factbook 2007*. Central Intelligence Agency, Office of Public Affairs, August 2007.

[3] K.K. Aggarwal, Yogesh Singh, Pravin Chandra, and Manimala Puri. Bayesian regularization in a neural network model to estimate lines of code using function points. *Journal of Computer Sciences*, 1(4):504–508, 2005.

[4] A.J. Albrecht and J.R. Gaffney. Software function, source lines of code and development effort prediction: a software science validation. *IEEE Trans. Software Engineering*, 9(6):639–648, 1983.

[5] Alan J. Albrecht and Jr. John E. Gaffney. Software function, source lines of code and envelopment effort prediction: a software science validation. pages 137–154, 1993.

[6] Giuliano Antoniol, F. Calzolari, L. Cristoforetti, Roberto Fiutem, and Gianluigi Caldiera. Adapting function points to object-oriented information systems. In *CAiSE '98: Proceedings of the 10th International Conference on Advanced Information Systems Engineering*, pages 59–76, London, UK, 1998. Springer-Verlag.

[7] Phillip G. Armour. Beware of counting loc. *Commun. ACM*, 47(3):21–24, 2004.

[8] Jayatirtha Asundi, Rick Kazman, and Mark Klein. Using economic considerations to choose among architecture design alternatives. Technical Report CMU/SEI-2001-TR-035, Software Engineering Institute, Carnegie Mellon University, December 2001.

[9] Rami Bahsoon and Wolfgang Emmerich. Archoptions: A real options-based model for predicting the stability of software architecture. In *Proceedings of the Fifth ICSE Workshop on Economics-Driven Software Engineering Research*, 2003.

[10] Rami Bahsoon and Wolfgang Emmerich. Applying archoptions to value the payoff of refactoring. In *Proceedings of the Sixth ICSE Workshop on Economics-Driven Software Engineering Research*, pages 66–70, 2004.

[11] Rami K. Bahsoon. *Evaluating Architectural Stabilty with Real Options Theory*. PhD thesis, University College London, October 2005.

[12] Sushil Krishna Bajracharya, Trung Chi Ngo, and Cristina Videira Lopes. On using net options value as a value based design framework. *SIGSOFT Softw. Eng. Notes (EDSER)*, 30(4):1–3, 2005.

[13] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA, 1999.

[14] Carliss Y. Baldwin and Kim B. Clark. *Advancing Knowledge and the Knowledge Economy*, chapter Between "Knowledge" and "The Economy": Notes on the Scientific Study of Designs, pages 299–328. MIT Press, 2006.

[15] Michel Benaroch. Managing information technology investment risk: A real options perspective. *Journal of Management Information Systems*, 19(2):43–84, Fall 2002.

[16] Michel Benaroch and Robert Kauffman. Justifying electronic banking network expansion using real options analysis. *MIS Quarterly*, 24:197–225, 2000.

[17] Fischer Black and Myron S. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–54, May-June 1973.

[18] Barry Boehm. *Software Engineering Economics*. Prentice Hall PTR, October 1981.

[19] Barry Boehm. *COCOMO II Model Definition Manual*. University of Southern California, 2000.

[20] Barry Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark Ellis Horowitz, Ray Madachy, Donald Reifer, and Bert Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall PTR, 2000.

[21] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost models for future software life cycle processes: Cocomo 2.0. *Annals of Software Engineering*, 1(1):57–94, December 1995.

[22] David W. Cearley, Jackie Fenn, and Daryl C. Plummer. Gartner's positions on the five hottest it topics and trends in 2005. http://www.gartner.com/DisplayDocument?doc_cd=125868, 2005. Last accessed on Jan. 21st, 2008.

[23] Computer Associates International. Ca-estimacs. http://www.ca.com/products/estimacs.htm. Last accessed on May 21st, 2006.

[24] John C. Cox, Stephen A. Ross, and Mark Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7(3):229–263, 1979.

[25] Joseph A. Dane. Modular program size counting. Master's thesis, University of Hawaii, December 1999.

[26] Department of Defence Data and Analysis Center for Software. Modern empirical cost and schedule estimation tools. http://www.thedacs.com/techs/estimation/comparison.shtml. Last accessed on Jan. 21st, 2008.

[27] Hakan Erdogmus. Comparative evaluation of software development strategies based on net present value. In *ICSE'99 International Workshop on Economics-Driven Software Engineering Research (EDSER1)*, May 1999.

[28] Hakan Erdogmus. Valuation of complex options in software development. In *ICSE'99 International Workshop on Economics-Driven Software Engineering Research (EDSER1)*, May 1999.

[29] Hakan Erdogmus. Value of commercial software development under technology risk. *The Financier*, 7, 2000.

[30] Hakan Erdogmus. Valuation of learning options in software development under private and market risk. *The Engineering Economist*, 47(3):304–353, 2002.

[31] Hakan Erdogmus, John Favaro, and Michael Halling. *Value-Based Software Engineering*, chapter Valuation of Software Initiatives Under Uncertainty: Concepts, Issues, and Techniques, pages 39–66. Springer Berlin Heidelberg, 2006.

[32] Hakan Erdogmus and Jennifer Vandergraaf. Quantitative approaches for assessing the value of cots-centric development. In *METRICS '99: Proceedings of the 6th International Symposium on Software Metrics*, page 279, Washington, DC, USA, 1999. IEEE Computer Society.

[33] John Favaro. A comparison of approaches to reuse investment analysis. In *ICSR '96: Proceedings of the 4th International Conference on Software Reuse*, page 136, Washington, DC, USA, 1996. IEEE Computer Society.

[34] John M. Favaro, Kenneth R. Favaro, and Paul F. Favaro. Value based software reuse investment. *Annals of Software Engineering*, 5:5–52, 1998.

[35] F.R. Freiman and R.E. Park. Price software model-version 3: An overview. In *Proceedings, IEEE/PINY Workshop on Quantitative Software Models, IEEE Catalog No. TH0067-9*, pages 32–44, October 1979.

[36] Galorath Inc. Seer-galorath estimating center. http://www.galorath.com/present_estimat.html. Last accessed on Jan. 21st, 2008.

[37] Handy Heffner and Larry Fulton. Topic overview: Service-oriented architecture. http://www.forrester.com/go?docid=38503, May 2006. Last accessed on Sept. 23rd, 2007.

[38] Handy Heffner and Larry Fulton. Topic overview: Service-oriented architecture. http://www.forrester.com/Research/Document/Excerpt/0,7211,42528,00.html, June 2007. Last accessed on Jan. 21st, 2008.

[39] Israel Herraiz, Gregorio Robles, and Jesus M. Gonzalez-Barahon. Comparison between slocs and number of files as size metrics for software evolution analysis. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 206–213, Washington, DC, USA, 2006. IEEE Computer Society.

[40] John C. Hull. *Options, Futures and Other Derivatives*. Prentice Hall, June 2005.

[41] International Function Point Users Group (IFPUG). Function point counting practices manual 4.2.1, January 2005.

[42] R.W. Jensen. An improved macrolevel software development resource estimation model. In *Proceedings, ISPA 1983*, pages 88–92, April 1983.

[43] C. Jones. *Programming productivity*. McGraw Hill, 1986.

[44] Caspers Jones. Backfiring: converting lines of code to function points. *Computer*, 28(11):87–88, November 1995.

[45] Rick Kazman, Mark Klein, and Paul Clements. Atam: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, August 2000.

[46] Chris F. Kemerer. An empirical validation of software cost estimation models. *Commun. ACM*, 30(5):416–429, 1987.

[47] Barbara Kitchenham. The problem with function points. *IEEE Sostware*, 14(2):29–31, 1997.

[48] Jo Ann Lane. Factors influencing system-of-systems architecting and integration costs. Technical Report USC-CSE-2006-614, Software Engineering Institute, Carnegie Mellon University, September 2006.

[49] Robert M. Leitch. Assessing the maintainability benefits of design restructuring using dependency analysis. Technical Report TR03-04, University of Alberta, Department of Computing Science, December 2002.

[50] Steven A. Lin. The modified internal rate of return and investment criterion. *The Engineering Economist*, 21(4):237–247, January 1976.

[51] Graham C. Low and D. Ross Jeffery. Function points in the estimation and evaluation of the software process. *IEEE Transactions on Software Engineering*, 16(1):64–71, January 1990.

[52] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, and Rebekah Metz. Reference model for service oriented architecture 1.0. Technical report, OASIS, August 2006. Committee Specification 1.

[53] Paul P. Maglio, Savitha Srinivasan, Jeffrey T. Kreulen, and Jim Spohrer. Service systems, service scientists, ssme, and innovation. *Commun. ACM*, 49(7):81–85, 2006.

[54] J. E. Matson, B. E. Barrett, and J. M. Mellichamp. Software development cost estimation using function points. *IEEE Transactions on Software Engineering*, 20(4):275–287, 1994.

[55] McKinsey & Company, Tim Koller, Marc Goedhart, and David Wessels. *Valuation: Measuring and Managing the Value of Companies*. John Wiley & Sons Canada, Ltd., 4^{th} edition, May 2005.

[56] Emilia Mendes. *Cost Estimation Techniques for Web Projects*. IGI Publishing, 2008.

[57] Robert C. Merton. Theory of rational option pricing. *The Bell Journal of Economics and Management Science*, 4(1):141–183, Spring 1973.

[58] NationMaster.com. Nationmaster. http://www.nationmaster.com. Last accessed on Jan. 21st, 2008.

[59] Ipek Ozkaya, Rick Kazman, and Mark Klein. Quality-attirbute-based economic valuation of architectural patterns. Technical Report CMU/SEI-2007-TR-003, Software Engineering Institute, Carnegie Mellon University, May 2007.

[60] Robert E. Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR-20, Software Engineering Institute, Pittsburgh, PA, 1992.

[61] Ser-Huang Poon and Clive W. J. Granger. Forecasting volatility in financial markets: A review. *Journal of Economic Literature*, 41(2):478–539, June 2003.

[62] PRICE Systems LLC. True's and price s: Software development and lifecycle estimating models. http://www.pricesystems.com/products/true_s_price_s.asp. Last accessed on May 21st, 2006.

[63] L.H. Putnam. A general empirical solution to the macro software sizing and estimating problem. *IEEE Transactions on Software Engineering*, pages 345–361, July 1978.

[64] QSM Inc. Slim-estimate. http://www.qsm.com/slim_estimate.html. Last accessed on Jan. 21st, 2008.

[65] H.A. Rubin. A comparison of cost estimation tools. In *Proceedings, ICSE 8*, pages 174–180, August 1985.

[66] Software Productivity Research. Spr document catalog. http://www.spr.com/catalog/. Last accessed on Jan. 21st, 2008.

[67] Software Productivity Research. Spr knowledgeplan. http://www.spr.com/products/knowledge.shtm. Last accessed on Jan. 21st, 2008.

[68] Eleni Stroulia and Robert Leitch. Understanding the economics of refactoring. In *Proceedings of the Fifth ICSE Workshop on Economics-Driven Software Engineering Research*, 2003.

[69] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference*, pages 99–108, New York, NY, USA, 2001. ACM Press.

[70] Albert Tarantola. *Inverse Problem Theory and Methods for Model Parameter Estimation*. Society for Industrial and Applied Mathematics, 2005.

[71] Jean Tessier. Dependency finder. http://depfind.sourceforge.net/. Last accessed on Jan. 21st, 2008.

[72] University of Southern California - Center for Software Engineering. Codecount. http://sunset.usc.edu/research/CODECOUNT/. Last accessed on Jan. 21st, 2008.

[73] Bas van der Raadt, Jaap Gordijn, and Eric Yu. Exploring web services from a business value perspective. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 53–62, Washington, DC, USA, 2005. IEEE Computer Society.

[74] Yi Wen. The business cycle effects of christmas. *Journal of Monetary Economics*, 49:1289–1314, September 2002.

[75] Gio Wiederhold. What are web services worth? http://www-db.stanford.edu/pub/gio/2005/WebWorth1.pdf, February 2005.

[76] Gio Wiederhold. What is your software worth. *Communications of the ACM*, 49(9), September 2006.

[77] Rama Yelkur and Maria Manuela Neveda DaCosta. Differential pricing and segmentation on the internet: the case of hotels. *Management Decision*, 39(4):252–261, 2001.

Appendix A

Acronym Reference Guide

Included for convience is a list of expansions of acronyms used in this thesis, loosely grouped by subject matter.

SOA Related

- BPEL: Business Process Execution Language
- SOA: Service-oriented Architecture
- SOAP: Simple Object Access Protocol[1]
- UDDI: Universal Description, Discovery, and Integration
- WSDL: Web Service Description Language
- WSIL: Web Service Inspection Language
- XML: eXtensible Markup Language

Economics

- GDP (OER): Gross Domestic Product (Official Exchange Rates)
- GDP (PPP): Gross Domestic Product (Purchasing Power Parity)

Software Engineering Economics

- ATAM: Architecture Tradeoff Analysis Method
- CBAM: Cost Benefit Analysis Method
- COTS: Commercial Off-The-Shelf (software)
- DEMI: Delivered Executable Machine Instructions
- DSI: Delivered Source Instructions

[1] With version 1.2 of the SOAP standard SOAP lost its acronym status; the complete name of the protocol is simply SOAP.

- DSM: Design Structure Matrix
- FP: Function Point
- IFPUG: International Function Point Users Group
- KSLOC: Thousands of SLOCs
- SLOC: Source Line of Code

COCOMO Family of Models

- COCOMO: Constructive Cost Model
- COCOTS: Constructive COTS (Commercial Off-The-Shelf) Model
- CORADMO: Constructive Rapid Application Development Model
- COSOSIMO: Constructive System-of-Systems Integration Cost Model

COCOMO Parameters

- CM: Percent of Code Modification
- DM: Percent of Design Modification
- IM: Retest Impact, as a percentage

Value Models

- CDF: Cumulative Distribution Function
- DCF: Discounted Cash Flow
- DSM: Design Structure Matrix
- IRR: Internal Rate of Return
- MIRR: Modified Internal Rate of Return
- NOV: Net Option Value
- NPV: Net Present Value
- NPVI: NPV Incentive
- PI: Profitability Index

ROMAN Framework

- ROMAN: Real Opions-based Modularity ANalysis
- RomanDSS: ROMAN Decision Support System